

TRI FILE COPY

ESD-TR-72-121 Vol. 1

TRI Call No.

75758

Volume 1, # 2.

Copy No.

1 of 2 Cyls.

A STUDY OF FUNDAMENTAL FACTORS UNDERLYING  
SOFTWARE MAINTENANCE PROBLEMS: FINAL REPORT

20 December 1971

ESD RECORD COPY

RETURN TO

SCIENTIFIC & TECHNICAL INFORMATION DIVISION

(TRI), Building 121Q

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS  
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)  
L. G. Hanscom Field, Bedford, Massachusetts 01730

Approved for public release;  
distribution unlimited.

(Prepared under Contract No. FI9628-71-C-0125 by Corporation for  
Information Systems Research and Development/CIRAD, 401 N. Harvard,  
Claremont, California 91711.)



AD739479

#### LEGAL NOTICE

When U. S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

#### OTHER NOTICES

Do not return this copy. Retain or destroy.

ESD-TR-72-121

A STUDY OF FUNDAMENTAL FACTORS UNDERLYING  
SOFTWARE MAINTENANCE PROBLEMS: FINAL REPORT

20 December 1971

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS  
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)  
L. G. Hanscom Field, Bedford, Massachusetts 01730

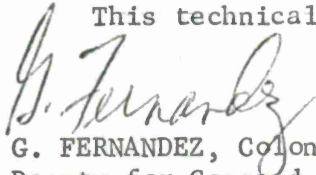
Approved for public release;  
distribution unlimited.

(Prepared under Contract No. F19628-71-C-0125 by Corporation for  
Information Systems Research and Development/CIRAD, 401 N. Harvard,  
Claremont, California 91711.)

## FOREWORD

This effort was conducted in support of Project 2801, Task 280119 by CIRAD, Inc., 401 N. Harvard, Claremont, California under Contract F19628-71-C-0125. The work reported here was accomplished during the period February through December 1971. It represents the first part of a continuing investigation into problems of maintenance programming. This effort was monitored by Dr. J. B. Goodenough, ESD/MCT.

This technical report has been reviewed and is approved.

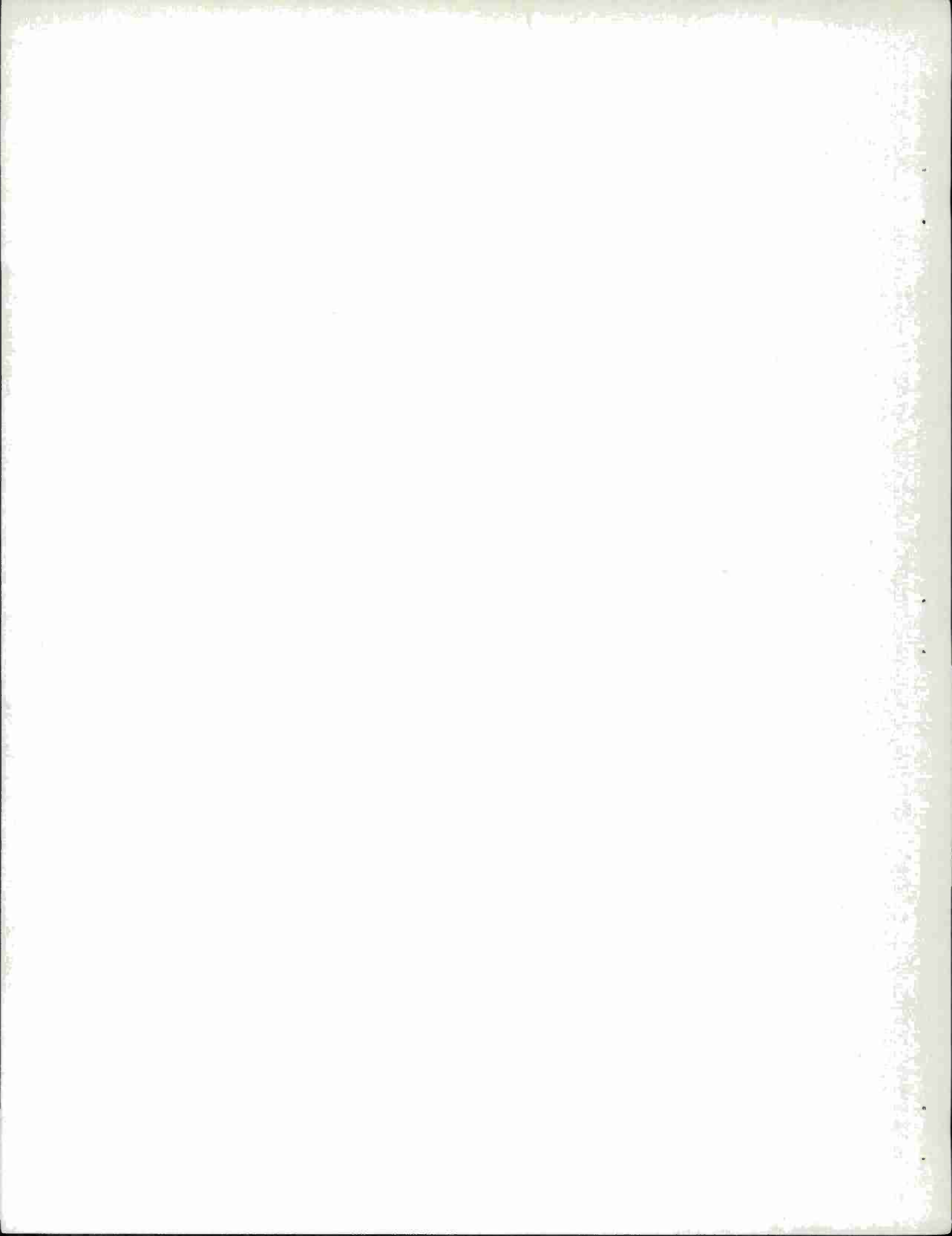


G. FERNANDEZ, Colonel, USAF  
Deputy for Command & Management Systems



## ABSTRACT

"Problems faced by programmers who must maintain programs someone else wrote" were identified. They were reduced to three fundamental inhibiting factors: (1) the limited rate at which people can make "relevance tests," (2) over-confirmation in clues required before hypothesis-testing, and (3) human vulnerability to distraction and procrastination. Studies suggested collectively by these factors were conducted. The studies (1) ascertained that programmers tend to think in terms of conceptual groupings whose objective identification would be helpful, (2) indicated that it was feasible to trace the path the programmer takes as he prepares to make a modification, and (3) identified a few tentative measures of the degree of maintainability of computer programs.

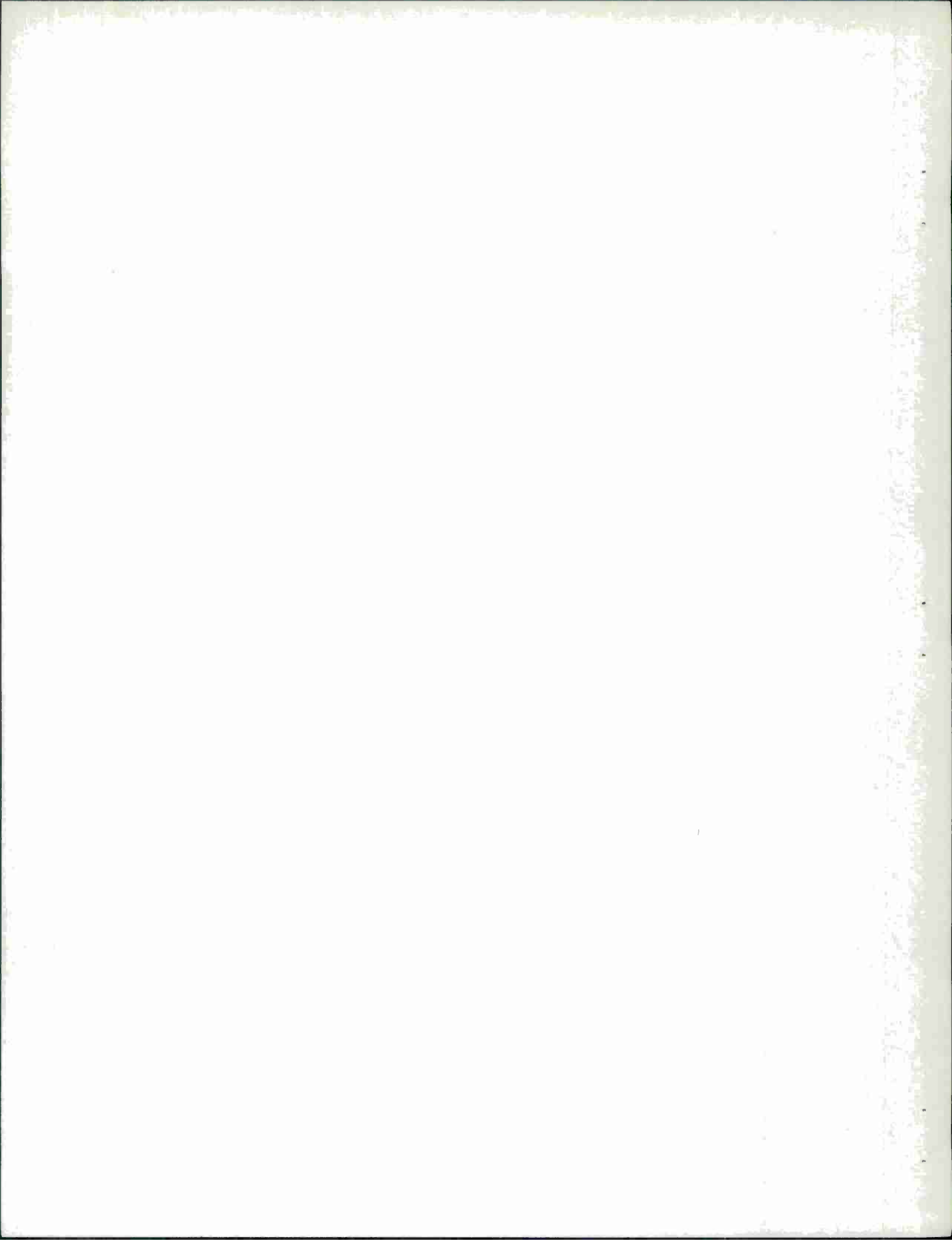


## TABLE OF CONTENTS

	<u>Page</u>
PREFACE: ORGANIZATION OF THIS VOLUME	1
PART I - REPORT ON STUDIES OF PROBLEMS	3
SECTION 1 - INTRODUCTION AND SUMMARY OF PART I	5
1.1 Introduction to Part I	5
1.2 Study Phase Procedure	6
1.3 Summary of Major Conclusions	7
SECTION 2 - PROBLEM IDENTIFICATION AND ILLUSTRATION	11
2.1 Preliminary Identification of Problems	11
2.2 Preliminary Conclusions	18
2.3 Additional (Logic-Oriented) Conclusions	20
2.4 Charts of Inhibiting Factors	24
SECTION 3 - REDUCTION TO FUNDAMENTAL FACTORS	39
PART II - RECOMMENDED EXPERIMENTS AND STUDIES	41
SECTION 4 - WEIGHTING FACTORS, AND SOME PROPOSALS	43
4.1 Weighting Factors	43
4.2 Projects of Less Than Top Priority	46
SECTION 5 - TOP-PRIORITY PROJECTS	71
5.1 Selection Criteria	71
5.2 Metrics for Maintainability	72
5.3 Conceptual Groupings	75
5.4 Path Analysis Feasibility Study	77



PART III - REPORT OF EXPERIMENTAL STUDIES	81
SECTION 6 - OVERVIEW OF EXPERIMENTAL RESULTS	83
6.1 Principal Findings	83
6.2 Overlap in Findings	85
6.3 Discussion and Recommendations	86
SECTION 7 - PATH ANALYSIS FEASIBILITY STUDY	89
7.1 Introduction	89
7.2 Objectives of the Experiment	89
7.3 Procedure	90
7.4 Results	91
7.5 Recommendations	96
SECTION 8 - METRICS FOR MAINTAINABILITY PILOT STUDY	101
8.1 Introduction	101
8.2 Objectives of the Study	101
8.3 Procedure	102
8.4 Results	103
8.5 Recommendations	113
SECTION 9 - CONCEPTUAL GROUPINGS PILOT STUDY	115
9.1 Introduction	115
9.2 Method of Study	116
9.3 Results of the Study	118
9.4 Discussion	134
9.5 Recommendations	136
REFERENCES	139

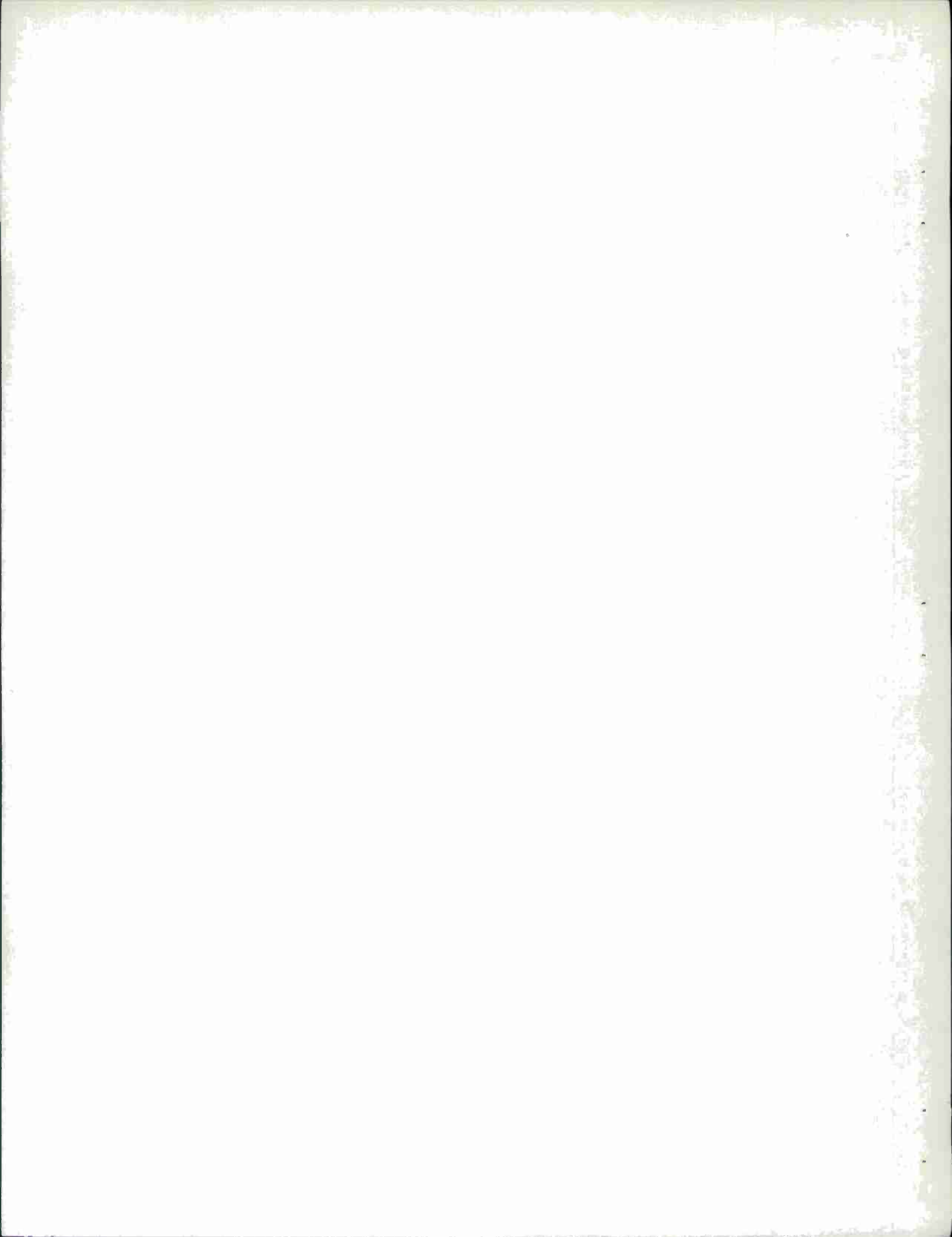


## APPENDICES

(Separately Bound)

- I Statement of Work for this Study
- II Maintenance Programming: Questions and Answers
- III BUIC Case Studies and Diary
- IV A Study of Factors....at Chrysler
- V CSA Software Maintenance Report
- VI Scenarios and Questions; Higher Order Language and Maintainability
- VII Technical Approach and Aims/Path Analysis Feasibility Study
- VIII Instructions for Path Analysis Experimental Programmer
- IX Specification of Modifications for Path Analysis
- X Minimum Paths According to Staff Programmer
- XI Experimental Programmer's Verbalizations while Making Modifications
- XII Guidelines for Keeping a Magnetic Tape Log of Program Maintenance Mental Processes
- XIII Tape Logs of Experimental Programmers
- XIV Language Statement Types Which Define Conceptual Groups





PREFACE: ORGANIZATION OF THIS VOLUME

"....Contractor will focus on problems faced by programmers who must maintain programs someone else wrote." That quotation, from the Statement of Work (Appendix I), indicates the focus with which this report was begun.

The report grew over the span of nearly a year's work. Findings have been integrated into one editorial unit in this volume, cast as much as possible in the objective terms of a scientific paper. But the chronology of discovery may still be seen.

The findings unfold in the three distinct parts of the report:

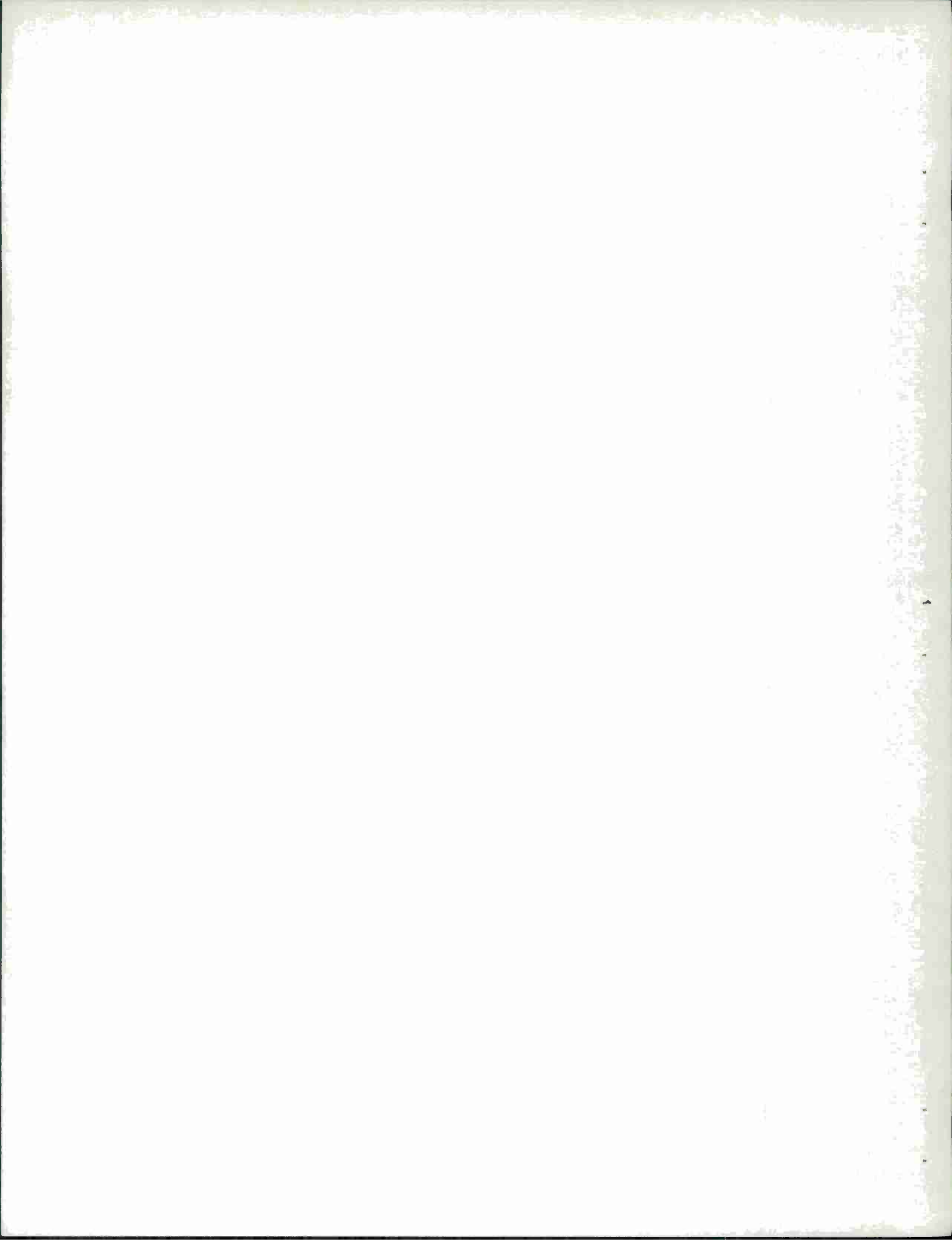
Part I explores the "problems faced by programmers," and finally reduces those problems to the "fundamental factors" inhibiting the programmers' efficiency.

Part II proposes a number of further studies and aids which might help solve the problems just analyzed. Criteria, which hopefully are realistic because they are based on case studies and the empirical analyses of Part I, are applied to these proposals. Finally, top-priority projects are identified.

Part III reports the execution of those projects, and the results obtained from them.



PART I  
REPORT ON STUDIES  
OF  
PROBLEMS



## SECTION 1 - INTRODUCTION AND SUMMARY OF PART I

### 1.1 Introduction to Part I

Part I of this report is an account of a study of problems of maintaining computer programs: of difficulties in modifying and correcting programs which make up a fairly large software system, and which other programmers have written.

To the programmers interviewed in this study, the problem is clear: Maintenance programming is unpopular and difficult. But that statement carries little information. Exactly what is difficult about maintenance programming? Why is it difficult? The right questions need to be asked, and then answered.

In the opinion of the authors, a valid approach to the right questions is the following: There are interactions between (1) the programmer, (2) the program, and various "representations" of it, and (3) the computer and physical environment. What aspects of these interactions constrain the efficiency of the programmer? What are the fundamental factors behind these constraints?

This approach excludes some legitimate but separate considerations. For example, management may unwisely assign a programmer to the wrong job. But if management is totally wise in its personnel assignments, maintenance programmers will still face situations which lower their efficiency. It is these situations which lower their efficiency. It is these situations which are the subject of this report.

## 1.2

### Study Phase Procedure

The experience with computers of each of the authors spans about a decade. A dilemma immediately arose: This experience should not be wasted, but neither should it be allowed to bias the results of the study. To overcome this dilemma, the authors' experience was augmented by bringing in the lessons said to have been learned by several other persons in different installations at different times.

The broadening of the experience base (which was considered as raw data) was done in the following ways:

1. Questionnaires were prepared and administered to several people who had what seemed to be good experience in maintenance programming and/or direction of that programming. Results are presented in Appendix II, "Maintenance Programming: Questions and Answers."
2. A special report on maintenance programming at Chrysler Corporation was obtained. It is Appendix IV, "A Study of Factors Inhibiting the Performance of Maintenance Programmers at Chrysler Corporation."
3. A similar report, drawn largely from experience in the aerospace industry, was obtained from Computer Software Analysts, Inc. It is Appendix V, "CSA Software Maintenance Report."
4. Concrete examples of maintenance problems with the BUIC software system were collected. Collection was accomplished in two ways: First, a senior BUIC programmer searched his and other's experience for case histories of typical maintenance problems. Second, another programmer kept a diary of the problems he encountered in maintaining the system. The results are presented as Appendix III, "BUIC Case Studies and Diary."
5. A search was conducted for published literature of the maintenance problem.

This experience base has served as the resource material for the analyses presented in this report.



At no one time were the "raw data" combined and reduced. Rather, they were "digested" as each "bite" came in. As the work progressed, significant confirmations began to emerge. For example, one author had observed that his programmers (on a large natural-language processing system) were not good detectives in that they needed more clues to locate the source of a bug than they ideally should have needed. Independently, the published literature showed that people's estimates of compound probabilities, as inferred from their behavior, differ grossly from the mathematical probabilities (e.g., Edwards, 1962; Cohen, 1960). This mis- or non-calculation by people was viewed as a factor behind the poor debugging detective work of the programmers. Also independently, an interviewee (Covill, in Appendix II) declared that the typical programmer does not form hypotheses well. He either "makes a great big detailed test of A, and learns that's not the answer," or he "leaps immediately to a conclusion and puts in a patch" which does not work. This declaration was taken as confirmation that programmers do not find it easy to draw anything approaching optimum solutions from probabilistic clues regarding locuses of failure.

As other confirmations developed, the research team sought to convert them into a description of the fundamental factors which inhibit the efficiency of maintenance programmers.

Section 2 details the steps in the process of maintenance problem identification. It presents illustrations of maintenance problems. Finally, it reduces the problems to fundamental factors inhibiting efficiency.

Section 3 describes several proposed solutions and areas in which research might be advocated. It also contains an evaluative system for estimating the feasibility of the research at this time.

Part II of this report details the selection procedure used to isolate three pilot studies.

### 1.3 Summary of Major Conclusions

From the analysis reported hereafter, the most basic principle which emerged was this: All fundamental factors inhibiting the efficiency of a maintenance programmer stem from areas of disharmony. A working arrangement links the programmer with the computer, with the software, and with the documentation and other tools. The non-human things can be configured in different

ways. The programmer, in general, cannot.

There are limits to human abilities. In physical activities, such as running a race, this fact is completely obvious. In cognitive areas, such as rapidly examining clues, the fact of human limitations seems often to be overlooked. Section 2 and Appendix B contain many examples of problems which would never have occurred if the maintenance programmer had been a "Super-Brain." But he was not. And the hardware and software were not configured to harmonize with human traits. So the disharmonious result was inefficiency and mistakes.

The lessons of Section 2 can be taken to show that the most general factor degrading maintenance efficiency is a lack of this quality which the authors have termed "harmony." More specifically, Section 2 seems to show that the inefficiency derives largely from failure to configure hardware, software, and documentation to conform to three human limitations. In decreasing order of importance, these limitations are:

1. The limited rate at which people can make what will be called "relevance tests." (For an example of a relevance test, imagine a programmer glancing at a piece of code, or at a clearly written commentary, to see if it is relevant to his maintenance assignment.)

Because of this human limitation, output routines impair efficiency when they require a programmer to scan too many things to find what he wants. Documentation creates disharmony when its needles of relevant information are hidden in a haystack of other statements.

One might conclude that the programmer needs "a relevant information finder." This conclusion, however, is only a re-statement of the problem. A more useful conclusion is the following: We need to know more about the conceptual groupings which a programmer can use to reduce the number of "straws" at which he has to look. This general thought will be expanded in Part II.

Also in Part II is a discussion of a study of exactly what is difficult about making reference checks. This study will help insure that later "inventions" meet real needs.

2. The extent of over-confirmation in clues which people require before they will test or accept a new hypothesis.

(For an example of over-confirmation, imagine a programmer seeing, in the span of a day's work, eight clues that a given routine was miscalculating; theoretically two clues should have sufficed, but the programmer did not examine the routine until he had seen the eighth clue.)

Because of the human tendency to require over-confirmation, today's documentation and related tools slow down problem-solving when they fail to display clues at the right times, in a good format, and in the proper conceptual units.

The projects proposed in Part II will help lay a foundation for harmony and efficiency in this area, particularly in that they imply that the future programmer, using the benefits of this research, will have to keep fewer conditional considerations in mind simultaneously.

3. Human vulnerability to distraction and procrastination.  
(For example, a programmer starts to document a change, hears someone else talk about another problem, gets interested, ... and the documentation is soon out of date.)

The computer has fewer senses than a person; it is less distractable. Therefore, it fails to contribute to harmony and to efficiency when it fails to do certain little tasks: those which it could do and which a person might not do. Record-keeping tasks are conspicuous examples.

Partly for this reason, Part II will propose an identification of some of the real tasks in which a maintenance programmer will probably engage sometime in the future.

The above words describe the seeds and the forest. The trees follow.





## SECTION 2 - PROBLEM IDENTIFICATION AND ILLUSTRATION

### 2.1 Preliminary Identification of Problems

During the first portions of this phase, two techniques were used to identify problems that grow out of effectiveness-inhibiting factors for maintenance programmers. The techniques were:

- a) A questionnaire was designed and administered to individuals with maintenance programming experience. One part of the questionnaire, entitled "Thoughts For The Future", addressed the question of what new tools, regardless of cost, the interviewees could think of that would, in their opinion, increase effectiveness.
- b) Members of the research team wrote scenarios of "ideal" maintenance programming environments that were as free of inhibiting factors as their imaginations would allow. These scenarios were based on the questionnaire results, subjective experience of the research team, and a search of the relevant literature.

Lists of questions were then derived from the scenarios. These questions were intended to make explicit the research areas that would have to be undertaken to achieve the environments described in the scenarios.

Appendix VI details the development of the scenarios and questions.

#### 2.1.1 The Questionnaire Results

Many illustrations of maintenance problems appear in the first three appendices to this report. Appendix A, giving the questionnaire results, is particularly interesting. Of those results, perhaps the most enlightening are the answers to the "blue-sky" questions. These results are given below.

Incidentally, the "blue-sky" questions were asked at the end of the interviews. Thus the interviewees had spent some time in addressing specific questions relating to the maintenance programming tasks before being asked the questions.

(Note: The following material is abstracted from Part VII of Appendix II.)

Question:

Now we'd like to turn to an entirely different way of looking at tools for program maintenance. If money were virtually no object, what sorts of tools would you ask to have designed, to make your job as easy as possible?

Answers:

- On-line Capabilities

- A way of keeping all of the current data base definitions and a dictionary available at object time to allow selective display (dump) of anything from one word to the whole program by symbolic reference on a terminal.
- Dynamic status of buffers available through time.
- Breakpointing of programs controllable from a terminal.
- Traces available on a console with options such as
  - tracing on jumps only
  - tracing only a specific piece of coding
  - tracing keyed to the setting of particular items or registers.
- Ability to elicit implicit references by displaying all areas of memory affected by a particular program.
- Ability to entable and display the last "n" values attained by specified objects of interest.
- Ability to run an old and new version of a program in "parallel" to pinpoint differences in behavior.

- Batch Oriented Debugging Aids
  - Ability to screen out extraneous information from dumps.
  - "Improved" set/use program with "convenient" formats.
  - A "model" of the system that would permit verification of completeness of inputs.
  - A way to isolate change patches to guard against
    - "adverse effects" on existing code, branches and item settings
    - inadvertent transfers into the patch.
- Documentation
  - Automatic updating of set/use tables.
  - Computerized updating of documentation.
  - Inclusion of dynamically discovered implication/interface information in the formal documentation.
  - Centralization of documentation to allow immediate access of current version to all concerned.
- Operations
  - Coordination and integration of maintenance procedures with installation operations.
  - Reduction in number and complexity of tasks required to factor a change into the system.
- Languages
  - Ability to compile code directly from a flowchart.



- Hardware

- Ability to reproduce specific hardware failures.

2.1.2 The Scenario Results

The second technique (after the questionnaire) of problem identification was the "Utopian scenario."

The method used in developing the scenario material was as follows:

- a) Lay out a scenario for a Utopian computer-aided software maintenance system.
- b) Break this scenario up into elementary sub-scenarios and for each such, develop a list of questions which would have to be answered before development of the facility implied in the sub-scenario could proceed.
- c) To each question assign one or more seriousness coefficients and develop a rational method for reducing a single score or a score vector for each sub-scenario out of the seriousness coefficients assigned to its questions.
- d) Use these scores to identify the questions which merit more prompt attention.

The scenarios are developed in detail in Appendix VI.

The scenario development was influenced by the work of E. W. Dijkstra (1968, 1969) and H. D. Mills (forthcoming) on hierarchical development and structured programming. Many of the concepts used here relate to their research on these topics.

Many questions were raised in developing the scenarios and in bringing them back toward reality. Those questions imply many of the problems that will have to be solved in working toward the maintenance programmer's Utopia.

In order to create an overview of the problems, the major questions are brought together here. An asterisk (\*) preceding a question indicates that it is one of the more appropriate research questions. The analysis which identified the appropriate questions is explained in Section 3 of this report.

The following questions were raised by the flowcharting scenario:

- \*FC1. Can the maintenance programmer (MP) work without hard-copy? Can he use only the cathode ray tube displays when examining and working with flowcharts? If he requires hard-copy, when and to what extent will he require it? When does the amount of hard-copy he might generate exceed the usefulness of the automated system, in particular, the "save" feature?
- FC2. What effect will out-of-date documentation in the file have on the MP's effectiveness? Statistically, how "out-of-date" might a file be at any one time compared to manual hard-copy files? What is the benefit that can be ascribed to the difference, if any?
- FC3. Is it possible to define a "unit" of flowcharting logic that will allow the MP to access displays that are defined for him and the system consistently? Should a unit defining algorithm be subjective (i.e., dependent on the particular system) or objective (i.e., independent of the system and thereby requiring some degree of conformity to a standard procedure for flowcharting)?
- FC4. What hardware characteristics facilitate the flowcharting system? Are these characteristics available in existing hardware? If not, what are the specifications for new hardware designs and what are the cost/benefit tradeoffs? (Some examples of facilitating hardware characteristics; high speed buffers for address "saving", hierarchical file organization with hard-wired dynamic storage allocation, etc.)

The following questions were raised by the scenario regarding structured documentation:

- \*SD1. Will maintainability and modifiability be improved or degraded in a system which is programmed by the structured approach?
- \*SD2. How will the "units" and "levels" of documentation be defined? As in the previous scenario, should such definition be subjective or objective?

- SD3. How difficult will it be for a MP to learn to use the system? How will this offset any advantages of the system?
- \*SD4. Can an effective algorithm be derived that will perform the text translation necessary for "expando" and "meta-code" techniques?
- \*SD5. What about the hard-copy "trade-off" alluded to in the preceding scenario? (It's not difficult to imagine a system configured in a way that the MP has to deal with greater volumes of more poorly organized hard-copy than with present manual systems.)
- \*SD6. What about the effects of out-of-date documentation referred to in the preceding scenario?
- SD7. What hardware characteristics facilitate the structured documentation system?

The following questions were generated by the scenario on "Interactive Aids in a Dynamic Environment":

- IA1. For an interpretive system, what are the optimum dimensions (the size) of the code, and how many inputs are maximum for effective operation of the system?
- IA2. For a play-back system what is the definition of a "frame"? (This is similar to the units definition question.) What is the definition of "resolution" of the frames? (i.e., homogeneity of the levels.)
- IA3. For a play-back system, how would frame synchronization between different resolution levels be achieved?
- IA4. If the MP detects an error in the code while using these aids, how will he insert (patch) the code? Will the technique for error correction create non-integrated code sequences? (i.e., "jumpy" frames in the play-back context.)
- IA5. In the context of these aids, what is the meaning of "recompilation"? (Obviously conventional recompilation will be extremely expensive.) Is it possible to derive segmented compilers that perform structural integration

with the whole complex of documentation and operational systems without, at the same time, requiring total multi-level recompilation?

- IA6. What should the operational definition of "debugged" be? Can statistical measures be derived that will determine when a program's reliability achieves a pre-determined value?
- IA7. What techniques, existing or new, should be used to achieve system security? What problems, such as interference between programmers, will such techniques create?
- IA8. In the real-time environment how should the question of "resource-ownership" be resolved? How does this question affect interfaces with the MP as he performs maintenance modifications?

The following questions grew out of the paper on higher-level or higher-order languages:

- HL1. What is the optimum (from the standpoint of MP effectiveness) length for labels? What are the trade-offs generated by label length between programmer effectiveness and compiler efficiency?
- HL2. What is the "optimum" mix of direct hardware implementation, software interpretation and compiled object code in implementing an HOL? Which language features are best handled in the ways cited? Can and should the programmer be able to specify the mode of implementation? What are the trade-offs between MP effectiveness and compiler efficiency?
- HL3. Can (should) the HOL language be structured? If so, what problems are raised in progressing between levels when maintenance is to be performed? How would multi-level testing be handled (i.e., segmented recompilation vs. total systems regeneration).
- HL4. What systems will be necessary to interface maintainable HOLs (structured or conventional) with structured documentation systems? How will modifications to one or the other be handled?



- HL5. Should the lowest system language level for the HOL be microprogramming structured? If so, what advantages and disadvantages result?
- HL6. What hardware features should be specified for the maintainable HOL? How are these features related to application requirements?
- HL7. What real-time application considerations are raised by a maintainable HOL? How will the HOL interface with operating systems and resource ownership algorithms? What maintenance problems will message/transaction handling interfaces raise?

The following questions were raised by the scenario on testing problems:

- \*TP1. What are the smallest units of data/documentation that should be accessible to the MP through the use of examination commands? What are the trade-offs between unit size and MP effectiveness?
- \*TP2. Should Systems Test Language (STL) logic be imbedded in production systems in order to facilitate rapid execution or should it be imposed at test time?
- TP3. What interface problems will occur between STL and structured documentation and code? What provision should be made in structured systems to accommodate an STL language?
- TP4. Aside from comparisons, what arithmetic routines, if any, should be available to the MP?

## 2.2 Preliminary Conclusions

### 2.2.1 Problem Boundary Conclusions

Emerging from the techniques employed were three conclusions that relate to the constraints or operating boundaries of the problem:

2.2.1.1 The questionnaires and literature search indicate that very little research attention is being given to the question of maintenance programmer effectiveness. In contrast extensive

efforts are being directed toward such topics as language specifications, operating systems, and application coding routines. For example, Rosin (1969), in tracing the historical development of operating systems, clearly indicates that the "systems editing" concept for maintaining operating systems was a post priori consideration of the design of the systems.

2.2.1.2 Debugging in a maintenance task is a much more subtle process than debugging original code. This fact was observed by programmers interviewed in this study, and Green (1970) independently states that frequently "....it is easier to re-write a program than to modify a program written by someone else."

Maintenance programming is concerned with the revision, to some extent, of an original program design. The revision may be incompatible, from an efficiency standpoint, with the original design. Ideally it would have been more efficient to incorporate the change at the time the original design was executed. However, the need for the change was typically not known at that time. Moreover, due to time and resource constraints, complete redesign is not practical. Thus, the maintenance programmer is faced with the problem of imposing a new design on an existing one which may create integration bugs that exist only because of the design inconsistencies.

2.2.1.3 Most of the existing programming aids and tools are oriented toward original design and programming work. (For example, Modern Data (Falor, 1970) presents a comprehensive survey of programming aids in the form of commercially available program packages. Nowhere in the article is addressed the problem of aids for the explicit purpose of maintenance.) The maintenance programmer must employ development tools to accomplish his tasks. In many cases, this puts him in the analogous situation of a plumber having to use the capital equipment of a large pipe manufacturer in order to modify the plumbing system of a small house.

## 2.2.2 Some Specific Points and Clues

Other, more specific results provided preliminary indications of the problems that might form a basis for research:

2.2.2.1 There was a reasonable consensus among the interviewees and project staff on turn-around-time desires during maintenance

debugging: for simple, clerical-type "goofs", as fast as possible. For others, up to overnight.

Complicating the picture is the fact that turn-around-time has an effect on the complexity of debugging aids required for effective programmer performance. Jonsson (1968) asserts "A longer turn-around-time demands more effective and thus more complex (debugging) aids."

There was strong consensus on the number of simultaneous projects that were optimum for a maintenance programmer to handle, given overnight turn-around-time: 1 to 3 projects.

An important observation was that the people stating opinions on this subject tended to make a virtue of necessity, saying that the turn-around-time to which they have adapted is the best possible. In contrast, a recent report (Boehm, et al., 1971) claims that, at least in one situation, a five-minute turn-around-time was most productive but least popular.

2.2.2.2 The maintenance programmer often must "simulate" the computer in order to understand the results of a particular set of tests. (See, for example, the description of the "four classical steps in the debugging procedure" in Bernstein and Owens (1968)). Tools that would facilitate this process seem to be highly desirable.

2.2.2.3 The clerical functions of keeping track of counters, label designations, data locations, etc., form a large part of the maintenance task. Tools that would automate appropriate portions of this process seem to be desirable.

2.2.2.4 The problem of integrating static conditions as seen in code and dumps with the dynamic execution of the program is one that presents the maintenance programmer with many conceptual difficulties. For example, Miller and Maloney (1963), Baecker (1968), and Grishman (1970) discuss attempted solutions of the problem of integrating static debugging aids with the dynamics of program execution. Better techniques are needed for defining the limits of perception under these conditions, and for formalizing these definitions and automating those processes which are found to be appropriate for automation.

### 2.3 Additional (Logic-Oriented) Conclusions

Section 2.1 was essentially a reporting job. It spelled out the opinions and experiences of several "authorities." Section 2.2 synthesized the lessons learned from those exercises. The



present section (2.3) is more logic oriented. It examines some of the needs of a maintenance programmer who is looking for a bug, and it draws some logical conclusions.

### 2.3.1 Documentation Units Definition, and Focusing

Mills (1970) implies the existence of two programmer needs of great importance. Quoting from page 217 of the article (underlining is ours):

"We classify as a documentation unit any right hand side of a rule which reduces to one of the following syntactic entities...

SIMPLE STATEMENT

STATEMENT

DECL

PROGRAM

"When programmers make an informal effort to arrange their programs for readability they typically start each documentation unit, as defined above, on a new line and use indentation to correspond in a general way with syntactical nesting in the program. We recognize that the problem is a subjective one, but we give a syntax-defined listing algorithm which is believed to satisfy the intuitive intentions observed in informal programming efforts."

Mills is saying, in essence, that there is need for a better definition of a documentation unit within the limited context of what he is trying to do. Mills recognizes that the problem is subjective; a study of this kind of problem must involve semantics as well as syntax.

Mills seems to recognize such a need. The whole value of his algorithm hinges on whether or not it does indeed "satisfy the intuitive intentions...". In this regard, Mills says only that it "is believed" to do so. He gives no opinions of agreement or supporting evidence.

In summary, then, the first need is for a more meaningful automatic definition of documentation units within the level of a small program such as the one which Mills studied.

The second need can be inferred from the above quote and relates to focusing, and to what might be called "Structured Documentation." In beginning a new job, a maintenance programmer does not start at the level of the code Mills was analyzing. He starts at a higher level, involving an overview of the entire software system; he then works his way down to the points at which code must be changed. During this focusing process, he would be helped by a higher level of documentation than that which a pure syntax approach can generate.

The second need, then, is for a way of providing higher-level documentation for blocks of documentation units, so the focusing process can be more efficient.

### 2.3.2 Convergent Search Techniques

Formal search techniques that would allow convergence to the cause of program malfunctions or would identify the "right" descriptive information about a particular program segment out of a large set of descriptive data do not exist. Kocher (1969) re-enforces this assertion by presenting current debugging techniques that do not include automatic convergence devices. Obviously such techniques would be of great value to maintenance programmers.

Such techniques should exist. Consider, for example, the game of "20 Questions." It is compatible with the kind of strategies to which humans are naturally best adapted: Discrimination learning (e.g., reward vs. punishment, approach vs. avoidance, etc.) and successive approximations (i.e., the "shaping of behavior": a process through which most infants naturally go).

Programmers should be particularly comfortable with "20 Questions"; it is essentially equivalent to a binary search routine for finding an item in a file.

But "20 Questions" does not show up very often in the real-life activities in which people engage. Most especially, it is not used in the activity of total concern here: maintenance programming.

The reason is intuitively obvious to some people with whom the topic of "20 Questions" search techniques has not been raised. Covill (in Appendix II) declared that software systems are hard to maintain because people "write programs which are absolutely planar."

Planar programs are not divisible. It is not possible to ask, "Is the problem in the right or the left half of the program? Does it occur before or after Subroutine 4?" Without well-defined units, questions like these are meaningless. And it is not possible practically to play the natural and efficient "20 Questions" game.

There are other, more mundane and corollary problems. For example:

1. At the level where the units should be, flowcharts are less meaningful: Boxes are less homogeneous and the logic is less uniform.
2. Other kinds of documentation become more difficult because of the lack of clear referents: It's hard to know what you are pointing at.
3. It is much harder to keep statistics on the involvement of units in failures if the units are not well-defined. This precludes the use of straightforward and effective methods of improving reliability and maintainability.

Liskov and Towster (1971) illustrate the difficulties involved in proving a system (or parts thereof) correct when the system has not been designed with testability as one of the design criteria.

In short, "20 Questions" remains just a game because of the lack of well-defined programming units of approximately homogeneous size.

### 2.3.3 Conclusions

Three conclusions have emerged as a result of the work performed in creating this section. They are:

2.3.3.1 An adequate definition of a documentation unit does not exist at the present time. A better definition is needed to complement such techniques as structured programming. It would also facilitate a general development described in Appendix V: "Structured Documentation," the logical analogue in documentation to structured programming.

2.3.3.2 The problem of logical "focusing" seems to be generic to maintenance programming work. The maintenance programmer is faced with the task of proceeding from the general (top level documentation, narratives, verbal specifications, etc.) to the specific (object and source code, data formats, reference labels, etc.) in order to actualize the modification he is attempting. Along the way his path is apt to be strewn with obstacles in the form of misleading information, improper conclusions and distractions. The solution techniques employed by programmers leads to much perceptual and intellectual shifting between the various levels of detail, and within levels, causing his "view" of the problem to be badly out of focus much of the time. The implementation of a technique such as structured documentation might provide the maintenance programmer with the ability to change "focus" resolutions with greater ease and fewer distorted perceptions.

2.3.3.3 Most programs are "planar" in logical construction, making iterative search techniques that narrow down on "bug" causes difficult, if not impossible, to formalize. Structured programming and documentation could allow the use of such techniques.

## 2.4 Charts of Inhibiting Factors

As any student of semantics knows, communication takes place at different levels of precision. And it is important to know what the limits of precision are.

Those limits (regarding the topic of maintenance programming) were not known (at least to the research team) when the interviews of Section 2.1 were begun. Now, however, some notes of caution can be sounded. For example, the lack of well-defined perceptual units in programming make it hard to refer to the perceptual points at which programmers have difficulty.



Among the problems of definitions and semantics, "documentation" especially requires more exact definition than presently exists. Questions such as "What are the elemental units (if any) of documentation systems" need to be answered. Or, it might be better to subsume "documentation" under "communication." In any case, the various representations of fundamental factors suffer from the lack of either concise or universal meaning of the words employed.

This assertion seems amply supported by several literature sources. Mills (1970) presupposed the existence of a programming system wherein the unit definition was provided. Liskov and Towster (1971) present a proof of correctness theorem that could not be practically applied to most existing programs because of their lack of unit structure and resulting intractability to anything but exhaustive testing. Dijkstra (1968) presents a structured system that derives hierarchies defined by the abstraction of systems functions without addressing the problems of correlating this structure to later modifications that might present abstractions of the abstractions.

Wirth and Weber (1966) encounter a similar problem of semantics. When discussing ALGOL 60 they state "The meaning of sentences ... was defined in terms of ordinary English with its unavoidable lack of precision."

A definition which is especially helpful in maintenance research is that of a "representation" of a software system. A representation is the system itself or any image, effigy, or likeness of it, no matter how crude the effigy may be. Given this definition, one can then isolate problems involving different representations of the system.

Following these notes on definitions are three charts of inhibiting factors. Each chart is derived from a different source of information; each is described in an introductory paragraph.

#### Chart 2.4.1 Summary of Answers

This is the longest and most significant of the charts. It organizes and summarizes a majority of the Phase One comments pertaining to inhibiting factors. The sources of these comments were the answers to the questionnaire, augmented by sources not covered in the other two charts.

In conformity with the authors' original approach, the chart is organized to group the comments in terms of interactions. The programmer interacts with himself, and with others; he also interacts with three representations of the programming system: the documentation, the program per se, and various "windows" or clue-generating displays of the system. This makes a total of five interactions. Each is covered on one of the following sub-charts (a through e) of Chart 2.4.1. (When a summary comment pertains to more than one interaction, it is included on more than one sub-chart.)



Chart 2.4.1.a Interactions of Programmer With (a) Himself

1. Now "the programmer has to simulate a computer," which is difficult for a human to do.
2. Experience with one software system can be generalized significantly to a different one.
3. Memory of "the gory details" fades quickly.
4. Can keep only about 1 to 3 projects active and outstanding at one time.
5. The better maintenance programmers are those who can "iteratively form and test hypotheses" efficiently.

Chart 2.4.1.b Interactions of Programmer with (b) Others

1. "Non-talkers" do not work well with "talkers."
2. Informal interchange is sometimes the most valuable source of information. Formal meetings sometimes just "sanctify" decisions previously made informally.
3. The need for maintenance tends to be left out of budgets.
4. Bull pens impair efficiency. Also, "desks aren't enough; you need a little work table for every 2 or 3 people. And ... blackboards."
5. Needed is "some way to integrate this whole thing with Operations."
6. "The user tended to think in terms of the application, with little concern for (the difficulty of) the programming effort."

Chart 2.4.1.c Interactions of Programmer with (c) the Program,  
Including Modifications Thereof

1. "The main constraint is (the time and work in) finding the right places to start ..."
2. Raising level of language sufficiently improves "sensibility" but introduces ambiguities.
3. There is very little cross-indexing between different representations (including dumps), and different levels.
4. Fast turn-around-time is desirable, at least for simple clerical "goofs", and possible during all stages of modifications.
5. Needed: "Flowcharts which cover all possibilities..." regarding paths within the computer, and point out (a) impossibilities (b) ambiguities.
6. A core dump is useless if the needed portion is "OUT ON DISC."
7. "There isn't any way to document an ill-conceived program that will make it maintainable.
8. Set/Use listings are helpful but should be automatically updated and should have more conditional control features.
9. "The second main constraint is being able to know what program your change will affect."
10. "Odd combinations" slip past the testing procedure.

Chart 2.4.1.d Interactions of Programmer with (d) Documentation, Including Requests for Change

1. "The main constraint is (the time, and work in) finding the right places to start ..."
2. "Narratives usually are written even worse than manuals."
3. There is very little cross-indexing between different representations (including dumps), and different levels.
4. Physical limits are inconvenient, especially in cross-indexing.
5. Needed: "Flowcharts which cover all possibilities" regarding paths within the program, and point out (a) impossibilities (b) ambiguities.
6. More urgent requests place more demands on human memory.
7. "Any flowchart (should be) complete on one 8-1/2 x 11 sheet." But a box can be the next sheet.
8. "There isn't any way to document an ill-conceived program that will make it maintainable."
9. Documentation often
  - a. is not accessible
  - b. is out of date
  - c. contains highly dispersed information
  - d. fails to tell "why." (And yet it could.)
10. Set/Use listings are helpful but should be automatically up-dated and should have more conditional control features.
11. "The second main constraint is being able to know what programs your change will affect."
12. Work ... in setting up and maintaining "fan-out" files.

Chart 2.4.1.e Interactions of Programmer with (e) Displays and Other "Windows"

1. "The main constraint is (the time and work in) finding the right places to start ..."
2. Track-and-trace features are valuable.
3. There is very little cross-indexing between different representations (including dumps) and different levels.
4. Fast turn-around-time is desirable, at least for simple clerical "goofs", and possible during all stages of modifications.
5. Needed: "Flowcharts which cover all possibilities..." regarding paths within the program, and point out (a) impossibilities (b) ambiguities.
6. "... usually got a full memory dump, and threw all of it away except 3 or 4 pages."
7. Core dump is useless if the needed portion is "OUT ON DISC".
8. A terminal "just makes it happen faster."
9. Needed is "a system to store all the current data base definitions, with the OK names for them... and a terminal to look at them."
10. Set/Use listings are helpful but should be automatically up-dated and should have more conditional control features.
11. "The second main constraint is being able to know what programs your change will affect."
12. Windows do not "tell how this garbage originated."
13. Needed might be "an automated information retrieval system, where you can easily call out any part of the history of this program."

Chart 2.4.2     CSA's List of Factors

This list of fundamental factors was generated by Computer Software Analysts, Inc., a sub-contractor (Appendix V). It represents a logical grouping of fundamental factors derived through the experience of the CSA staff members.



Chart 2.4.2     CSA's List of Inhibiting Factors

In chronological order, sources of difficulty are:

Change Request

1. Lack of knowledge of system operation.
2. Lack of knowledge of system components and organization.
3. Poverty of documentation.
4. Obsolete documentation.
5. Access to documentation.
6. Knowledge of application area principles.

System Change Design

1. Poverty of documentation.
2. Obsolete documentation.
3. Access to documentation.
4. Knowledge of system organization and components.
5. Knowledge of intersystem effects.
6. Identifying system components affected.

Program Change Design

1. Obsolete documentation.
2. Identifying documentation required.
3. Access to documentation.
4. Lack of detailed knowledge about individual system components.

### Program Code Development

1. Identifying code to be modified.
2. Lack of up-to-date listings.
3. Constraints in coding.
4. Access to the machine.
5. Updating documentation after modification.
6. Modifying data, allocation, structure, and content.
7. Reconstructing system after modification.

### Program Debugging

1. Generation of test inputs.
2. Decoding test outputs.
3. Determining successful system operation.
4. Isolating errors for correction.

### Testing and Validation

1. Development of system exerciser.
2. Reducing data from system exercise.
3. Determining successful system execution.
4. Determining level of testing required for system validation.

#### Chart 2.4.3     Chart of Chrysler Factors

This chart displays the fundamental factors derived from a study of the Chrysler Corporation computer systems organization (Appendix IV). It was based on the experience of the former manager of that organization. It groups the factors according to associated programming functions and provides weighting and ranking of seriousness of effect.

Chart 2.4.3.a Inhibiting Factor Score and Percentages\* By Individual Inhibiting Factors

Summary of Scores and Percentages.

1. Inhibiting Factors

<u>Factor</u>	<u>Score</u>	<u>% of Total</u>
Documentation	10.0	8.7
Machine Time	8.5	7.4
Verbal Communication	8.4	7.4
Work Surroundings	8.0	7.0
Prod. Env. Test	7.0	6.1
Consultation/Supv	6.8	6.0
Application Knowledge	6.0	5.2
Progr. Structure Knowledge	6.0	5.2
Inter Systems Effects	5.8	5.0
Machine Displays	5.4	4.7
Prog. Languages	5.2	4.5
Audit Procedures	4.4	3.8
Operations Practices	4.2	3.7
Prod. Responsibility	4.2	3.7
Clerical Duties	4.0	3.5
Test Data Missing	4.0	3.5
Audit Practices	3.6	3.1
Production Diagnostics	3.6	3.1
Results Review	3.0	2.6
Operations Org./Procds.	2.8	2.4
Production Bugs	2.4	2.2
Design Changes	1.2	1.1

\*From Chrysler Study, Appendix IV

Chart 2.4.3.b Inhibiting Factor Scores and Percentages\* (b)  
by Groups of Factors

<u>Group</u>	<u>Score</u>	<u>% of Total</u>
Inadequate Knowledge	30.8	26.9
Inadequate Org./Prcdrs.	30.0	26.1
Poor Communications	20.8	18.4
Inadequate Environment	16.5	14.6
Missing Aids	16.4	14.2

\*From Chrysler Study

Chart 2.4.3.c Inhibiting Factor Scores and Percentages\* (c)  
by Work Activities

<u>Activity</u>	<u>Score</u>	<u>% of Total</u>
Production Shakedown	37.2	32.4
Debugging	23.2	20.3
Quality Assurance	13.8	12.0
Test Design	10.9	9.4
Initial Request	9.0	7.8
Diagramming & Coding	8.2	7.1
Systems Analysis	7.8	6.7
Program Planning	5.4	4.7

\*From Chrysler Study



### SECTION 3 - REDUCTION TO FUNDAMENTAL FACTORS

The charts in Section 2.4 represent a summary of the findings of this study. But that summary still fills several pages with scores of assertions. This section performs a further summation; it reduces the charts and other conclusions to what seem to be the really most Fundamental Factors.

Loosely speaking, any summary has to be done within a certain field. The field used here will be broad.

Narrowly speaking, one could say that cars need better carburetors... or that computers need better dump routines. More broadly, one could say that more efficient transportation is needed... or that better documentation is needed. Most broadly, one could claim that we need more efficient ways of putting people into close communication with each other... or that we need more efficient communication between the programmer and all representations of the program.

Now a major purpose of this study is to identify the most valuable areas for future research. So the broadest field will be used in summarizing the inhibiting factors.

According to Chart 2.4.1, "The main constraint is (the time and work in) finding the right places to start." In other words, the maintenance programmer has to look at too many things to find out which are relevant to his problem. Similarly, according to Chart 2.4.3, "inadequate knowledge" appears as the most important group of inhibiting factors. Furthermore, "lack of knowledge" (and, by implication, the need to take many steps to discover the relevant knowledge) shows up repeatedly in Chart 2.4.2.

From these summaries the following important conclusion is reached: The major factor inhibiting maintenance programmer performance is failure to compensate in some way for the limited rate at which people can make what we call "relevance tests."

This conclusion is reinforced by survey papers in the literature (Overton, 1961; Overton, 1968) which show that man's "throughput rate" in making such tests is indeed limited.

Phrases like "inadequate knowledge" and "lack of knowledge" are ambiguous. They could refer to ignorance of a basic datum, or they could refer to failure to "put two and two together" and

come out with the right conclusion. In this connection, Chart 2.4.1 points out that the better maintenance programmers are those who can "iteratively form and test hypotheses" efficiently. That chart also complains that "windows" do not "tell how this garbage originated;" i.e., they do not help the programmer combine facts from different sources into a consistent theory of what is happening. Finally, the literature (Edwards, 1962) shows that a fundamental human weakness is the relative inability to deal with the theoretical implications of compound probabilities.

From these and similar findings a second important conclusion is reached: A second major inhibiting factor is failure to compensate in some way for the extent of over-confirmation in clues which people require before they will test or accept a new hypothesis.

Chart 2.4.1 reports a general opinion when it states that bull pens impair efficiency. The literature includes books (e.g., Broadbent, 1958) examining the fact that "signals" are lost when a person's attention is diverted by another "communication." Chart 2.3.1.3 says that "inadequate environment" in general accounts for 16.5% of the inhibiting factors, and that "work surroundings" in particular rank fourth in a list of 18 causes of trouble.

These reports support a third major conclusion: A third major inhibiting factor is failure to compensate in some way for human vulnerability to distraction and procrastination.

The following sections of this report suggest and propose some research related to these three Fundamental Factors.

PART II  
RECOMMENDED  
EXPERIMENTS AND STUDIES



## SECTION 4 - WEIGHTING FACTORS, AND SOME PROPOSALS

### 4.1 Weighting Factors

Two criteria will be used to help evaluate the desirability of proposed research projects. One criterion is the extent of connection with the Fundamental Factors. A second, and more quantitative, criterion is similarity to questions which analysis has shown to merit prompt study.

#### 4.1.1 Analysis of Questions

The questions posed at the end of the scenarios (Appendix VI), when compared with the fundamental factors inhibiting programmer effectiveness, will support the rationale for research plans. In order to organize and analyze the effect of these questions it was necessary to first impute to them a measure of their relative importance. Relative importance is determined by two factors; the time and resources required to supply answers to the questions, and, the importance of the answers to resolution of fundamental factors that inhibit the effectiveness of maintenance programmers.

In developing the rankings given in Section 2.1 of this report, the two scores were added for each question and numerical ranks assigned on the basis of the sum-scores.

The scale that was used to evaluate these questions is two-part: a) The inhibiting factors score (S1) is based on a scale that ranges from 0 to 10. The questions were scored subjectively. The higher the score, the more relevant is the answer to the questions in removing inhibiting factors. A score of 10, for example, indicates that the answer to the question would definitely remove or eliminate one or more factors that significantly inhibit the effectiveness of maintenance programmers.

b) The time/resources score (S2) is based on a scale ranging from 1 to 9. These scores were also derived subjectively. The higher the score, the lower the resources and elapsed time required to supply answers to the indicated question. It is assumed that time and resources are roughly equivalent which, at this level of estimating, seems to be a reasonable assumption. S2 scores relate to estimated elapsed time and resources as shown in the following table:



	SCORES								
	9	8	7	6	5	4	3	2	1
EST. ELAPSED TIME (years)	1	1	2	2	2	3	3	3	4*
EST. RESOURCES (in man years)	1	2	2	3	4	4	5	6	6*

\*or more

#### 4.1.2 Question Scores

The following table reviews the questions in the sequence they were written and indicates the scores for each. Assuming that inhibiting factors and time/resources have equal weight in determining selections for research, the scores can be added to get a sum which indicates the value of promptly answering the questions. Asterisks mark the questions with the eight highest sums.



Chart 4.1.2 Question Scores and Sums

<u>Question</u>	<u>Score</u>		<u>Score</u>
	<u>S1</u>	<u>S2</u>	<u>Sum</u>
<u>Flowcharting Scenario</u>			
1. *Hard-copy	6	9	15
2. Out-of-date documentation	10	2	12
3. Unit definition	9	4	13
4. Hardware	6	1	7
<u>Structured Documentation</u>			
1. *Maintenance optimization	10	8	18
2. *Unit/level definition	10	6	16
3. Learning difficulty	3	2	5
4. *Translation algorithm	8	7	15
5. *Hard-copy	7	9	16
6. *Out-of-date documentation	10	4	14
7. Hardware	8	1	9
<u>Interactive Aids</u>			
1. Optimum dimensions	6	7	13
2. Frame/resolution defin.	8	4	12
3. Frame synchro.	6	3	9
4. Error correction	8	3	11
5. Recompilation	9	2	11
6. Definition of debugged	4	6	10
7. System security	6	7	13
8. Resource ownership	4	5	9
<u>Higher-Level Languages</u>			
1. Label length	5	8	13
2. Direct/indirect compiling	7	5	12
3. Structured HOL	9	4	13
4. HOL/Structured interface	8	2	10
5. Microprogramming	6	5	11
6. Hardware	8	2	10
7. Real-time considerations	7	2	9
<u>Testing Problems</u>			
1. *Examination unit size	8	6	14
2. *STL imbedded logic	6	8	14
3. STL/Structured interface	8	3	11
4. Arithmetic routines	4	8	12

#### 4.2      Projects of Less Than Top Priority

This sub-section contains proposals for studies and experiments which are not recommended for inclusion in the present activities, i.e., they are of less than top priority.

The proposals contained here, along with the top-priority recommendations which follow, comprise action-oriented conclusions derived from the various analytical and study efforts of the project.

##### 4.2.1      Recommended Experiments

The recommendations are organized as follows:

1.    Introduction and Rationale: a brief description of why in relation to the fundamental factors, an area is worthy of study and a description of relevant background information.
2.    Devices: a description of the physical apparatus to be used in the experiment.
3.    Procedure: the general steps to be followed in conducting the experiment.
4.    Analysis: The relevant areas of the experiment to be examined, and how they are to be examined and conclusions drawn.
5.    Priority: a recommendation, in narrative format for the timing of the work described in relation to its importance and to other recommendations.

1. Experiment: Modifiability of Conventional vs Structured Approach

Structured programming and documentation (Mills, 1970; Dijkstra, 1969) may be an effective technique for facilitating the maintenance programmers' need to make relevance tests and to confirm hypothesis. If this conjecture is true, then a maintenance programmer should be able to modify a structured program faster and with greater ease. However, this is a hypothesis which has not been tested. Accordingly, a test is proposed of the relative difficulty of making extensive modifications in (1) a conventionally-designed software system, versus (2) one programmed strictly according to the structured approach.

This experiment would be of value because it would provide answers to scenario question 1, under structured documentation, concerning maintenance optimization.

Devices

The basic device used in the experiment would be a programming system derived from the initial specifications for a software system of moderate complexity: e.g., a payroll system.

Also used in the experiment would be a collection of requests for changes in the system: Say, the equivalent of two years' requests for modifications in the system. These would reflect developments such as the introduction of a credit union, adding payroll deductions for state income taxes, adding employee options regarding deductions for federal income taxes, changing social security laws, and removing a company savings plan.

Procedure

Experimental subjects would be four people, or four groups: Groups 1a and 1b would initially flowchart the system by the conventional approach.

Groups 2a and 2b would initially flowchart it using the structured approach.

Groups 1a and 2a would then install, in the conventional flowcharts, all the requested modifications.

Groups 1b and 2b would then install, in the structured flowcharts, all the requested modifications.

The reason for subdividing groups 1 and 2 is to segregate (1) the effects of the programmer's initial design experience from (2) the effects due to the flowcharting approach per se.

### Analysis

Main interest would be focused on the comparison of the relative modifiability of the systems developed according to the two different approaches. This would be measured by comparing Groups 1a and 1b with Groups 2a and 2b, in terms of time required, reports of difficulties encountered, etc.

Other interesting interactions could also be studied. These would relate to programmer experience (before modification) and flowchart approach.

Of tertiary value would be data on the relative difficulty of doing the initial design by each of the two approaches.

### Priority

The structured approach to programming and documentation may offer significant improvements in maintenance programmer effectiveness; if not, other solutions must be developed. In either case, work cannot proceed until the questions to be answered by this experiment are resolved. Therefore, high priority is placed on the execution of this experiment.

The effort required for this experiment is estimated at 4 to 6 man-years. This precludes its inclusion in Phase Two of this project. Then this study should follow that of "Units/Level Definition."

In Part II of this report a study entitled "Metrics for Maintainability" is recommended for inclusion in the experimental phase. It addresses some of the points raised above which should first be investigated.



## 2. Experiment: Microcosmic Test Bed

Two of the 8 most important questions emanating from the scenarios dealt with the need that the maintenance programmer had for hard-copy in order to effectively perform program modifications. A corollary question is: How does the use of an interactive console and structured documentation modify the need for hard-copy?

The questionnaire results (Appendix II) and Chrysler Study (Appendix IV) raise the question of how informal interchange between programmers affects the maintenance programmer's efficiency.

This experiment is designed to provide data for answering these questions.

### Devices

A complete, miniature, but complex for size software system and its documentation will be constructed in a hierarchial, structured manner.

The entire source code and documentation structure will be contained on 100 35mm slides to be mounted on a 100 slide carousel projector with push-button controls allowing fast back-up and advance.

### Procedure

A group of programmers is given a set of prototype questions from which at least two examinations will be selected.

They are then individually "turned loose" on the carousel for a prescribed period of time. Programmers in Group B are given a pad of paper and told to take any notes which they desire. Programmers in Group B are relieved of writing implements and papers and specifically forbidden to take notes.

Each programmer is given an examination immediately after the learning session, and the time and accuracy of answers is recorded.

The same experiment (with new groups of naive subjects) would then be performed with one group forbidden to talk with each other, and the other group given the time and opportunity to discuss with each other what they have learned.

### Analysis

The first examination results will be compared to determine the efficacy of hard-copy (as represented by programmer notes) in learning about the system.

The second set of examination results will be analyzed to determine the effect of informal interchange on the learning process.

### Priority

This experiment is beyond the scope of the experimental phase of this project. Its cost is estimated at 2 to 4 man-years. It should be given consideration for follow-on work (i.e., not among the first projects, but immediately thereafter).



### 3. Experiment: Key-Word Sifter

One of the problems confronting a maintenance programmer in the performance of his work is his inability to readily find specific sections of documentation that relate to a question he has about the functioning of the system. The questionnaire (AppendixII) and Chrysler report (AppendixIV) in the appendix of this document illustrate the fundamental inhibiting factors that reside in the programmer's inability to readily "lay hands on" pertinent documentation. Scenario question 3 under Structured Documentation also raises this problem.

A facility not presently available to maintenance programmers, that might substantially alleviate this problem, would be a "key-word sifter." In effect, this device would allow the programmer to input key-words or phrases that referred to areas he had questions about and would output the information relevant to that area. A corollary feature might be the ability of the device to accept an information area designation and to output a dictionary of the key-words associated with that area.

This experiment proposes to study some of the characteristics of such a device.

#### Devices

A relatively small programming system would be selected along with several modifications to be made to the system.

#### Procedure

Three experienced programmers, (A, B and C) would be involved in the experiment.

Programmer A would be given all of the system documentation and code and would make the modifications requested working in a conventional manner.

Programmer B would be shown the system documentation and would have the requested modifications described. He would then construct a list of topics that he would like further information about in order to complete the modifications. From this list would be drawn, with B's concurrence, a list of key-words. He would then attempt, with only that documentation so referenced, the modification, making note of any areas he needed further information about that were not available through the list. After this

attempt, a new list would be constructed and the documentation referenced made available. Any references not used would be deleted. This phase would proceed in this manner until the modification was successfully completed.

Programmer C would then, using the final reference list and material constructed by B, attempt the modification from scratch..

### Analysis

The times taken for the modification by programmers A and C would be compared. The number of iterations required by programmer B would be analyzed to determine the cost and feasibility of constructing such a system. The difficulties encountered in getting the right information at the right time by A and C would be compared to determine the facility of key-word referencing as opposed to conventional methods.

### Priority

This experiment is too ambitious to be included in the experimental phase, since its cost is estimated at 1 to 2 man-years. However, the proposed "sifter" might prove to be a valuable tool to the maintenance programmer, and the cost of the experiment is certainly not prohibitive compared to the value that might be derived. Therefore, it is recommended that it be included in the early part of the on-going research program.

#### 4.2.2 Recommended Studies

The recommendations are organized as follows:

- 1) Introduction and Rationale: a brief description of why, in relation to the fundamental factors, an area is worthy of study and a description of relevant background information and the hypotheses.
- 2) Approach: the method to be taken in examining the hypotheses and/or formulating new ones.
- 3) Expected Results: the tangible results expected to be produced by the study.
- 4) Priority: a recommendation, in narrative format, for the timing of the work described in relation to its importance and to other recommendations.

##### 1. Study: Unit/Level Definition

One of the highest ranked questions in the preceding section of this report was the one dealing with Unit/Level Definition. Consistently, the raw data supporting this report alludes either directly or indirectly to the efficiency inhibiting problems created by the lack of consistent Unit/Level definitions. See, for example, the questionnaire, (Appendix II), BUIC Case Studies, (Appendix III), and Chrysler Report, (Appendix IV); and Dijkstra (1969, 1968) and Mills (1970, forthcoming).

In "looking down on" any level of information, one often attempts to arrange individual items into blocks or units or "chunks" (Miller, 1956); then one thinks about the chunks or units. At the assembly language level, for example, one may group one sequence of four commands together as a limit test, and then think about the limit test as a logical unit in its own right. Or, one may group another set of commands together because they perform a well-defined function, and then pay attention to that function. But, regardless of how it is done, the point is generally valid: In looking at any level, people usually attempt to group things.

Disharmony and attendant lack of perception necessary for effective understanding occur when information is arranged in a way that retards the discernment of such groupings or promotes the discernment of ambiguous groupings.

Assuming that units of programming information can be defined that are unambiguous and understandable by the programmer, a second question arises: In what ways should the units be grouped or classified together to reflect the various levels of information available? The units obviously interact with the level of detail involved. They will be different in COBOL than in a detailed flowchart, and different in the latter than in an intermediate-level flowchart, and so on. Therefore, a study is proposed to seek answers to the following two questions:

- 1) How does one determine the degree of abstraction, or conversely, the degree of detail which should be included and excluded in a given level of programming information to make that level homogeneous to the logical perception of the maintenance programmer?
- 2) At a given level, how does one group basic information elements into larger units which are reasonably homogeneous, meaningful, and recognizable to the maintenance programmer?

#### Approach

In an attempt to get a reasonable limit for the size of an optimum unit, information theory will be applied to such things as code statements, and then the results will be related to published literature on human information-holding capacities.

At least at the level of a higher-level language, examination of the relationships with the characteristics of the language will be performed.

Following the general approach of semantics and target recognition studies, the general characteristics of clues which enable people to recognize meaningful units, will be identified.

Also examined will be the possible conflict between internal organizations (within a block) which are most understandable to a person versus those which are most efficient for a computer.

In an attempt to work toward usable definitions of homogeneous levels, concepts relating degree of abstraction to objective actions and/or interactions by people and/or computers will be developed. Existing literature will be reviewed. For example, Dijkstra (1968), in constructing a hierarchical multi-programming system, attempts to define hierarchy levels that correspond to abstractions of system functions.



Examples of hypotheses which might be examined are the following:

1. Each level of description or documentation or programming is really intended for a different audience.
  - a. At the lowest level -- that of the compiled object deck which has been translated or assembled into machine language -- the "audience" is the computer itself.
  - b. At the highest level, the audience is the user, or perhaps the customer who employs the user.
  - c. At the intermediate levels, the audience may be the same person -- e.g., the programmer -- but he may play different roles; i.e., he may act like a systems analyst in one role, like one level of programmer in another, and even like a compiler in another.
2. Each level serves two audiences and two purposes:
  - a. For its own level, it tells what to do and how to do it.
  - b. But, for its own level, it does not tell why. The why of what will be done is often found in the next higher level.

One approach, then, is to (1) define the roles which different people and the computer play at different times, and (2) define the level of information needed for each role.

If the role is well-defined, then the level of information appropriate to it will be homogeneous for it.

Another possible approach might use the concept of degree of truth, or percentage of exceptions. That is, it might be possible to say that a top-level statement is only P percent true with respect to the most detailed and precise version of the same statement; or, conversely, that the top-level statement would

contain (100 - P) percent exceptions from the point of view of its most complete restatement.

This approach is viewed as less promising than the audience/roles approach described above. But, if the numbers in it could be derived in any meaningful way, they would provide a powerful set of criteria for homogeneity and efficiency.

### .. Expected Results

The study will provide tentative answers to the two basic questions: How does one determine the degree of abstraction, or, conversely, the degree of detail which should be included and excluded in a given level to make that level homogeneous? At a given level, how does one group elements into larger units which are meaningful, recognizable, and reasonably homogeneous?

A result which might be even more valuable will be the following: What concepts developed here should be used for asking the right questions for applied research on hierarchical systems in the future?

### Priority

Since the concept of Units/levels definition is fundamental to the construction of generally applicable structured programming and documentation systems this study is viewed as having high priority. Cost is estimated at 3 to 5 man-years.

In Part II of this report, a study entitled "Conceptual Groupings" is recommended for the experimental phase of this project. It addresses some of the more basic points raised above. This study might profitably follow that one by a few months.

### 2. Study: Sensory Integration

The report of a famous experiment in physiology was entitled "What the Frog's Eye Tells the Frog's Brain" (Lettvin, et al., 1959). It reported that the frog's eye is (in our words) "wired up" to detect objects about the size of flies, moving at about the speed of flies. The frog's eye responds much less well to less interesting (from the point of view of a frog) things. These have less survival value.



People have similarly evolved a tendency to perceive some-things better than others. As a prime example, pictorial displays are more easily and reliably perceived than are numerical data (Newman, 1966). In general, a problem arises because people have not evolved in an environment full of computers. They are not most naturally interested in, and able to perceive easily, the things (i.e., print splattered on big pages) which present computers quite frequently produce. Many of the scenario questions deal, to some extent, with this perceptual problem.

But the problem is more complicated; it is not just a matter of pictures versus numbers. Senses interact. Human activities interact. Doing mental arithmetic can literally distort a person's perception of a graph. Looking for a number will lower one's chances of detecting a geometric figure.

Bearing these facts in mind, it is clear that computer displays are typically not well adapted to people. This means that the maintenance programmer is typically not working efficiently in his most basic perceptions of displays.

This study is proposed to find ways of improving the perceptual efficiency of maintenance programmers.

#### Approach

First, a survey of the literature on the basic forms that people can most readily perceive would be done.

Next, a survey would be undertaken of the work that has been done on determining how various activities and stimuli interfere with the perception of basic forms.

Last, a study of what activities a maintenance programmer will typically engage in at a display unit versus the kinds of information available for presentation on the display will be done. (The method here would rely largely on observation and questionnaires directed at programmers engaged in display work.)

#### Expected Results

The following study results would be expected:

- 1) Statements of the activities and kinds of information related to the programmer's console interaction.

- 2) A summary report, with critiques, of the relevant literature on basic perception and interference.
- 3) The design of a combination of display forms which individually would be easily perceived and which collectively would be integrated in the sensorium of the programmer most efficiently for perceiving the fundamental action of a program.

### Priority

This study is too ambitious to be included in the experimental phase of this project. Its cost is estimated at 1 to 3 man-years. However, the concepts raised in this study description seem to be fundamental to the understanding and alleviation of factors inhibiting maintenance programmer effectiveness. Therefore, its early inclusion in follow-on work is recommended.

### 3. Study: Meta-Languages

Two of the major effectiveness-inhibiting characteristics of maintenance programmers (and people in general) identified in this report were the limited rate at which they make relevance tests and their resistance to the acceptance and testing of new hypotheses. Much of the documentation associated with a programming system reinforces these characteristics. The background data for this report (see Appendices) copiously reports the symptomology of documentation ills: The documentation is misleading because it is out of date or ambiguous, unclear and hard to understand, lacking syntactical consistency and semantic clarity, useless (as an aid to understanding system structure), etc. A major proposition in this report is that the structured approach to documentation would possibly help remove the causes of these ills. The missing bridge between these two (i.e., ills versus proposed cure) is a method for determining the efficacy of proposed solutions without having to construct a solution and submit it to exhaustive experimental testing. The purpose of this study would be to develop such a bridge.

### Approach

In formal logic (Church, 1956) primitive languages have been developed in order to (among other things) examine language characteristics and logical relationships in a context free of semantic ambiguity. Often, meta-languages have been constructed from the principles derived in primitive language research. One

of the purposes of meta-languages is to provide a conceptual bridge between primitive languages and human, everyday, language use.

What is proposed here is a use of these techniques in reverse. In many senses, higher-level computer languages and documentation are meta-languages, bridging the gap between machine code and the natural language of the human being. It is proposed to seek a primitive language, in formal logical notation, that would be fundamental to the higher-level languages and documentation. (As a lemma, it is thought that it can be shown that machine code is not fundamental to these languages.)

First, an analysis of documentation for its essential logical elements would be performed using the work of others and supplementing this with new analyses where necessary. Then, from the formulation of these elemental logical characteristics primitive symbology would be extracted. On the basis of this symbology and the logical analysis a propositional calculus would be derived a meta-language.

#### Expected Results

The discovery of such a language would permit the examination of the logical characteristics of higher-level language and documentation. Such criteria as completeness and consistency could be tested. Logical weaknesses could be detected and proposed language corrections derived and tested. Generic theorems, giving direction to major language elements, could be proposed and proven or disproven.

#### Priority

Certainly this project is too ambitious and abstract to be consistent with Phase Two objectives. Its cost is estimated at 4 to 6 man-years. It is felt that the concepts and questions raised here could form a significant part of the on-going research program.

#### 4. Study: Play-Back System

Balzer (1969) describes debugging and monitoring system called EXDAMS that incorporates "motion-picture displays." These displays allow the programmer to look at program information



as a function of execution time, whose rate of change the programmer may increase, decrease or stop.

One of the causes of the inability of maintenance programmers to make adequate relevance tests and form and test hypotheses is the difficulty in relating the static information (dumps, listings, etc.) he normally deals with, to the dynamics of program execution. Many of the scenario questions under Interactive Aids deal with this problem.

Therefore, a study is proposed to determine the feasibility of developing a system that would aid the maintenance programmer in perceiving the dynamics of program execution.

### Approach

The approach entails at least the preliminary design of a playback system.

The gist of such a system is that a tape file is created which contains trace and dump information as the code is executed. This tape is later played back as a slow-motion movie on a CRT screen with the user able to speed up, slow down and jump forward or backward in the flow of action. The user is provided with extensive and flexible facilities for masking, filtering and summarizing the vast and otherwise unintelligible information content of the movie's data base. These facilities are exercised through commands such as

- "Collect and display a table showing successive values of variable APRIME showing program module and line number where created."
- "Operate at high speed until ALPHA is greater than BETA then slow down for closer inspection."
- "Make a new film starting at APROG Line 1 and ending at BPROG Line 35 using GAMMA=0 and DELTA=-1.3"
- "Display the node and chord skeletal structure of the program providing identification of the nodes (branch points) and chords (linear code), and display at each chord the number of times it was executed (total or under specified conditions)."

The study will first analyze the characteristics of existing play-back systems (such as EXDAMS) to determine the advantages and disadvantages in their use and their general applicability in resolving maintenance programmer efficiency inhibiting factors.

Next, detailed specifications for an "ideal" play-back system will be created. Then, each of the features of the "ideal" system will be analyzed to determine:

1. If such features already exist in an implementable form.
2. The method for implementing such a system.
3. The cost, in time and resources, of such a system.
4. The utility of such a system in reducing the factors that inhibit maintenance programmer effectiveness.

A major part of this detailed specification will deal with expanding the list of commands available to the programmer, some examples of which have been given above.

#### Expected Results

The "product" would be a detailed implementation plan for creating a play-back system, together with its costs and advantages.

#### Priority

The study cost is estimated at 1 to 3 man-years. Since there are major questions as to the feasibility of such a system, it is recommended that it be given low priority in follow-on work.

#### 5. Study: HOL's and Maintenance Programming

A recent study (Knuth, 1970) attempted to discover quantitatively "what programmers really do." As an analog to that work, it is proposed that a study be performed to find out what maintenance programmers "really do" in performing a modification to a programming system. In particular, the proposed study would concentrate on specific issues concerning higher order languages (HOL's).

One of the sections of Appendix VI is concerned with HOL's. While none of the questions associated with that section as among the eight most important questions (per the analysis of Section 4.1) it is felt, nonetheless, that the over-all group of questions is of sufficient importance to serve as the basis for the present study proposal.

In a sense, the study described below can be regarded as a logical successor to the first study proposal of Part II of this report in that it attempts, in part, to make use of the concepts and measures relating to "distant referents" which are therein to be developed to analyze the maintainability promoting or inhibiting features of existing and proposed HOL's.

This study proposes to concentrate on the following specific issues concerning HOL's:

1) Language Features:

(a) Certain features of existing and proposed HOL's are specifically intended to ease the burden of developmental and maintenance programmers. These include

- compiler time DEFINE/macro facilities;
- conditional compilation directives;
- the ability to name and subsequently include sections of code or data base definition by reference to said name, thus centralizing the locus of future maintenance effort;
- the PARAMETER statement of UNIVAC's FORTRAN V which may be regarded as a special case of the first category on this list;
- the use of the ON statement group as in PL/1 for conditional debugging output;
- the ability to request the construction of cross-references and other aids at compile time.



- (b) Other features of HOL's not including those mentioned in items (2) - (4) below are intended for other purposes, but have an inadvertent positive or negative effect on maintainability.

## 2) Program Control Structures:

The approach of Mills and Dijkstra to structured programming or constructive proof of correctness calls for HOL's lacking the GOTO statement. A companion study (in this section) proposes to examine the effects of this approach on maintainability.

In a similar vein, the maintainability implications of other control structures needs to be explored. A partial list of such structures would include recursion, coroutines and asynchronous (parallel) processes (Fisher, 1970).

## 3) Source Input Structure

Existing HOL's exhibit two types of source input structure. The FORTRAN type of structure generally calls for a main program and zero or more separately compiled subprograms. (UNIVAC's FORTRAN V provides a minor partial exception in this regard.) The need for globally defined data base items and arrays in this environment gives rise to features such as blank or named COMMON sections and BLOCKDATA program elements. Language features such as the ability to name subsequently included sections are added to alleviate the maintenance problems engendered by the appearance of a given COMMON block in myriad subprograms. An intuitive feeling about this approach to source input structure is that for a given size of program, the "distance" (however this is eventually defined and measured) of distant referents is increased.

By contrast, the ALGOL (60) type of source input structure calls for one program will all of its subprograms (except for library programs) contained within it. The scope of data definitions is implied by the nesting of the domains of definition in the source order. An intuitive feeling is that this type of structure has beneficial effects as regards "distance" of distant referents; but a simple change to a small part of such a program requires a recompilation of the total program, whereas the former approach requires only that portion of the program which is in error to be recompiled.

## 4) Method of HOL Implementation

In general HOL's may be implemented in one or a combination of three distinct ways:

- Interpretatively: The source text is scanned and executed by an interpreter. An example is APL.
- Compilation: Compilers are, of course, common.
- Hardware or Microprogram Implementation: The source text is translated and reordered in a vastly more simple and quicker process than in full compilation. The result of this process is still relatively close to source form and is directly executable in the hardware, or by microprogrammed interpretation. Examples are SPL on the SPLM and SYMBOL on SYMBOL.

The EL1 language (Wegbreit, 1971) being developed at Harvard allows the programmer the choice of either interpretation or compilation; and, moreover, the compiler is callable at object time by the programmer.

The four areas of concern outlined above are not wholly independent of one another. For example, it was noted above that certain language features are introduced into FORTRAN to allow global data definitions and still others are introduced to promote the maintainability of the resultant mass of definition text. Of note also is that there may be interactions between program control structures and the issue of total vs partial recompilation. In the future, moreover, if hardware implementation of HOL's is widely practiced, the latter issue may be of little importance.

### Approach

A project member would observe a maintenance programming environment using an HOL as the principal language. The first two aims of these observations would be to

- 1) Discover and classify the types of error corrections and modifications actually made by maintenance programmers, e.g.
  - Faulty data declarations
  - Failure to dimension an array
  - Array subscripts out of bounds, etc.

- 2) Record the frequency distribution of the types of error corrections.
- 3) Measure the efficacy of compile and object time diagnostic messages in pinpointing the cause of trouble to the maintenance programmer.

The next step would be to analyze the efficacy or retardant value of the four HOL language areas listed previously in the light of the results of 1) and 2) and as germane to the particular language being used.

These steps should, if possible, be repeated in one or more similar environments, using HOL's, to isolate the effects of specific language differences upon the distribution of errors and modifications. In this connection, it would be very valuable to secure the cooperation of the SYMBOL project at Iowa State University (Chesley and Smith, 1971) to observe maintenance programming using a hardware-implemented HOL that has no declarations, variable field length operands and free-form manipulation of data structures.

Parallel to this observation-study, a literature search and review would be conducted in the areas delineated by the four issues cited above.

#### Expected Results

Assuming that the observations cover enough different persons in each environment to vitiate the effects of individual differences, these observations and the literature search would be analyzed to produce the following results:

- 1) A statistical profile of the types of errors corrected and modifications actually made by maintenance programmers cross-sectioned by isolatable language features.
- 2) An analysis of HOL features in terms of their effects upon maintenance programming.
- 3) A delineation of the best features to be included in a optimal-maintenance HOL.

### Priority

This study is felt to be important in its own right, and in complementing other recommended projects: The studies of "Metrics for Maintainability" and "Analysis of Future Situation" in Part II of this report, and the "Path Analysis" study in this section. It is too ambitious for inclusion in this portion of the project, since the cost is estimated at 1 to 2 man-years.

#### 6. Study: Analysis of Future Situation

##### Rationale

Four facts introduce the last recommended study:

1. The third of the most important Fundamental Factors was human vulnerability to distraction, and human proclivity to procrastination.
2. The interviews revealed repeated complaints about distractions and (separately) out-of-date documentation (which bespeaks procrastination). Even in the BUIC examples, where contracts required the strictest standards of documentation, there were cases in which utility programs and other tools were not kept up to date.
3. A major feature of the maintenance programmer's imagined Utopia was the elimination of documentation per se, and its incorporation into a software system which could be accessed via a graphics terminal.



4. Similarly, a question which was said to be of high merit for early study was: How much of a need does a programmer really have for hard-copy? How would he really behave in front of a Utopian terminal?

Considering only the above four facts, one might propose an interesting and valuable experiment in which the dependent variables were human distractability, and punctuality of work; the independent variable would be represented in different working environments. However, such an experiment is not now proposed. It would not be practical. We do not have a supply of Utopian terminals; besides, the experiment would be too expensive.

What is proposed is an exercise which can share some of the experience of the first two studies -- especially the one on relevance tests and maintainability. Also, it does touch upon distractability, procrastination, and the degree of need for hard-copy. What is proposed is a sort of task analysis of the behavior in which a maintenance programmer would probably engage at a display.

More precisely, the proposal is for the making of predictions regarding the extent to which maintenance programmers will actually use new console-oriented facilities at their disposal.

Such predictions will have obvious economic value: They will help guard against the wasteful development of expensive tools which no one will actually use.

#### Approach

In general, the procedure will be to compose a consensual picture of a future, console-oriented, maintenance facility; and then to predict how it would actually be used. Predictions will be based in part on analogous experience with existing facilities.

#### Task 1: Compose Picture

Two major sources will be used to draw out the "picture" of the facility. One will be an extension of the Utopian Scenarios (from Appendix V). The other will be literature (e.g., Engelbart and English, 1968; Joyce & Cianciolo, 1967) advocating extensions of graphic displays for a variety of tasks.

Concepts presented in such literature will be transferred, when appropriate, to the maintenance area.

Task 2: Predict Usage by Analogy

The various features of the future facility will be examined with the following questions in mind: To what present capabilities are these features analogous? How are the present capabilities actually used?

For example, present hardware/software systems offer excellent scan-and-sort capabilities. But Hamming (1970) observes: "... when I examine what the typical software person does, I am often appalled at how little he uses the machine in his own work. A nonexpert, with a very long FORTRAN program to convert to local use, wrote a simple FORTRAN program to locate all the input-output statements and all the library references. In my experience, most programmers would have personally scanned long listings of the program to find them and with the usual human fallibility missed a couple the first time."

More generally, the capabilities of FORTRAN are well-known. How extensively are these various capabilities used? A study of the actual behavior of FORTRAN programmers (Knuth, 1971) provides empirical data. By analogy with such data, predictions will be made regarding the probable usage of the features in the future, console-oriented, maintenance facility.

The experience of the pilot study in the first project will also be helpful. Structured programming will probably be common in the future. The researcher who did the programming in the pilot study can profitably be asked what display features would have helped him in this work.

Task 3: Predict Limits by Theory

It is assumed that there are limits on how hard a programmer will work in front of a console. In human factors jargon, there are limits on the "perceptual load" he will accept. There are limits at the rate at which he will "through-put" information.

A study of such limits, as they appear in other tasks, will cast light on the limits which may appear in maintenance programming. For example, suppose it is assumed that a programmer will tolerate no more of a perceptual load at his



console than he will in driving to work. Then literature on perceptual limits in driving (e.g., Cumming, 1964) can be used to help estimate the limits of his performance at work.

#### Expected Results

The report of this project will broadly describe the predicted behavior of a maintenance programmer with a future console. In addition, two lists will be developed: one list of those "Utopian" features which would probably be used and be helpful, and another list of those features which probably would not prove to be utilized.

It should be emphasized that the results of this study will not be definitive and complete, as would the results of a full-blown task analysis of an existing situation. Nevertheless, the study results will provide valuable criteria for evaluating proposed future developments.

#### Priority

The estimated effort for this study is 4 man-months of technical effort. Because its results would contribute to other other studies (such as sensory integration) and interact with them, this study should receive next-to-top priority in the follow-on work.



## SECTION 5 - TOP-PRIORITY PROJECTS

### 5.1 Selection Criteria

Quoting the Statement of Work (Appendix I), "a small set of problems, factors, and/or proposed solutions" will be studied more intensively during the terminal months of this contract. The choice of those problems, etc., is described here.

The choices had to meet the following criteria: They had to relate to the Fundamental Factors and they had to involve questions which merited prompt study. As a redundant check on these two criteria, the choices also had to be realistic in the sense that they could be directly related to real problems. (This last criterion is redundant because of the realism of the "raw data" from which the Fundamental Factors were reduced.)

Another mundane criterion was that the scope of the studies had to fit within the time and manpower constraints of the experimental phase of this project. In summary, the achievable results had to offer the promise of guiding future studies into more efficient and productive paths.

The three projects recommended here relate to (1) metrics and concepts for quantifying the maintainability of software systems, (2) the definition of conceptual groupings which will improve maintainability, and (3) a feasibility study to qualify Path Analysis as a valid object of future study.

The projects are described, in turn, below.

## 5.2        Metrics for Maintainability

### 5.2.1     Rationale

The rationale for the choice of this project derives from the following three facts:

1.    The most important fundamental factor turned out to be the limited rate at which people can make relevance tests.
2.    A question of very high merit turned out to be the relative measurement of the maintainability of structured programming.
3.    The BUIC Case Studies and Diary show repeated examples of the generation of bugs because adequate relevance tests were not made, and of slowness in finding them because many tests had to be made. (A homey example comes from the Diary on March 17, 1971: "Tom left work early today. His wife needed transportation home. The hang-up was caused by ... Tom had installed this EPC in a previous BUIC version and, in his attempt to shorten his job, he tried to use as much of the old code as possible" without fully examining its relevance to the present modification.)

The above facts lead to this conclusion: It would be good to be able to measure the rate and ease with which relevance testing is done; it would be good to have metrics which could be applied to programs to yield such measurements. The metrics could be used to compare the modifiability of structured programming with conventional programming. More broadly, they could help quantify maintainability in general.

The first project proposed for Phase Two, then, is the creation of the definitions and measures needed for the quantification of rate and ease of relevance testing.

#### 5.2.2 Procedure

The proposed procedure will entail the performance of the following tasks:

##### Task 1: Create Definitions and Measures

A search and analysis of the raw data gathered in Phase One (with probable emphasis on the BUIC Case Studies) will create a preliminary set of definitions and metrics applicable to relevance testing. In other words, the first task will be the development of concepts.

A concept which will be developed and used extensively is that of the distant referent.

A prime example of a distant referent is called (especially in COBOL) a "switch." One uses a switch (for example) when one reads in some data which can belong to any of n categories. What one does with a datum is determined by the category to which it belongs. One way of handling each datum is to set a switch by it to designate its category. Then, later, in the body of the program, one looks back at the switch to decide whether or not to handle a given datum at that point in the program.

Essentially the same example is called (especially in machine language) a "flag." One sets up a matrix, for example, examines each row to determine whether or not it falls in a certain category, and flags it if it does. Then, perhaps much later, one goes on to process only the rows bearing that particular flag.

Now it seems entirely logical to assume that a maintenance programmer has to do more work (make more relevance tests) when there are more distant referents, and when their distances are greater. Thus: The more distant referents there are, and the more distant they are, the harder a program is to modify.

Task 1 will include the more precise definition of "referent," some preliminary way of measuring "distance", and the development of such other, similar concepts as seem necessary and possible.



## Task 2: Perform Pilot Study

This task will be an exercise aimed at insuring that the concepts being developed in Task 1 (which may partially overlap Task 2) are applicable to the real world.

A computer program, already written in conventional style, will be selected according to the following criteria: It will be reasonably short, but it will contain several subroutines, and the logic will be rather complicated. The research team will either be familiar with the logic, or able to familiarize themselves with it rapidly. The program will be capable of considerable modification.

A program which may be selected is one in ALGOL which is well documented (Wersan, 1967). It performs a linear programming optimization under certain restrictions; for example, it must deal exclusively with integers.

The program will be rewritten in structured programming style.

The applicability of the concepts from Task 1 to the rewriting will be examined.

Modifications (for example, the removal of the integer-only and other restrictions) will then be specified for the program.

Using the concepts from Task 1, comparisons will be made of the relative difficulty of modifying the structured and the conventional versions of the program.

(An incidental benefit resulting from this task will be a very tentative, quasi-experimental answer to the question of the relative maintainability of structured versus conventional programs.)

The difficulties in using the concepts will be reported by both the programmer and an observer.

## Task 3: Refine and Recommend Concepts

Drawing largely from the experience of the pilot study, and from any relevant literature which may be found in the interim, the research team will refine the definitions and procedures used to quantify the rate and ease of relevance testing.

A set of concepts will be recommended to help quantify future analyses of maintainability.

#### 5.2.3 Manpower

The estimated effort for the study of metrics for maintainability is 4 man-months of technical effort.

### 5.3 Conceptual Groupings

#### 5.3.1 Rationale

The following three facts point to the second proposed problem area:

1. The second of the most important Fundamental Factors was the inability of programmers to utilize all of the information theoretically available in clues to resolve an ambiguous situation.
2. A very meritorious question was: What kind of size of "blocks" (of code or higher-level language) are most "visually graspable" and "semantically transparent" to the maintenance programmer?
3. Maintenance problems involving groupings which are not conceptually natural for people are illustrated in the BUIC Case Studies and Diary. (For example, on March 10, 1971, two groups of functions were intermixed and "the second problem was not noticeable while the first existed." And, on March 12, a similar problem arose because "this branch was not evident to the programmer searching the listing.")

The Mills approach to structured programming clearly recognizes the need for graspable units. To paraphrase Miller (1956), Mill's approach is "chunk-oriented."

Mills implies that good units must meet at least two criteria. First, they must not be too long. He declares (Mills, forthcoming, p. 9) that in a logic-oriented unit there may be "simply more than the eye can take in or the mind retain ...."

A second Mills criterion is more vague. It relates to the word "natural." He states (Mills, forthcoming, p. 11), "Each page (containing one segment) should represent some natural unit of

the program...."

Regardless of the particular criteria used in generating them, "good" units would have value. Fewer clues would be needed, in resolving fewer ambiguities about results of programming runs, if there were better conceptual groupings which were clearer to the programmer. Debugging could be done with more insight, and with less blind tracing. Therefore, another recommended problem for more intensive study is this: How can efficient conceptual groupings be defined, and what will foster them?

### 5.3.2 Procedure

In general, the proposed procedure is a search for contrasts -- for the characteristic differences between the conceptual groupings which people actually use and the more arbitrary groupings which may appear in documentation.

In particular, there will be three tasks:

#### Task 1: Collect Documentary Data

This task may prove to be impossible; the documentary data may be practically unavailable.

An example of coding in structured programming would be documentary data if the coding could also be automatically grouped syntactically by a procedure such as that of Mills' (1970). Then it would be possible to see what contrasts there are between the "chunk-oriented" structured programming units and the automatically-produced "documentation units."

#### Task 2: Collect Experimental Data

This task is essentially a little experiment. It consists of having a programmer take a program with which he is familiar and orally explain it to two members of the research team. Unknown to the programmer, the purpose will not really be to communicate the details of the program. The purpose will be to see what units the programmer actually uses in his explanation.

These units will be marked so that they can be contrasted with the units used in the formal documentation.

If task 1 proves to be impossible, the magnitude of Task 2 will be correspondingly expanded.



### Task 3: Analyze Data and Report

Analysis will consist of a search for discriminants: for the characteristics which differentiate between the "good" (chunk-oriented and orally used) units and the "not-so-good" units.

Of course, the distinguishing characteristics cannot be specified in advance. They might appear in terms of number of statements, number of do-loops, extent of change in function performed, or in any of many conceivable parameters.

The study report will emphasize the most significant characteristics which distinguish between "good" and "bad" units. Although the results of this rather brief study will be somewhat tentative, they will nevertheless be capable of helping the units used in future documentation be more compatible with those readily perceived by people.

#### 5.3.3 Manpower

The estimated effort for the conceptual grouping study is 3 man-months of technical effort.

#### 5.4 Path Analysis Feasibility Study

The maintenance programmer, as he attempts to modify a programming system, works in at least two intellectual dimensions. First, he is attempting to derive new coding structures that will produce the modification correctly and, second, he is, at the same time, attempting to learn enough about the structure and logic of the program to enable him to make the correct changes and to understand the cause of aberrations in the program that are the symptoms of wrong changes. There is a conflict between at least one kind of "learning", and "doing" (Wolin, 1962). This conflict produces a situation which may inhibit programmer effectiveness. One way of examining this possibility is through what will be called "path analysis."

Path analysis is based on the observation that any programmer attempting to modify a programming system follows a "path" through the documentation and code associated with that system. The first "step" on the path may occur when a programmer looks at a narrative for the system. Then he may proceed to an intermediate level flowchart and then to a section of code corresponding to the flowchart and then back to the narrative, and so on. At certain junctures in the path the programmer may "do" something, e.g., modify a section of code, or write a control procedure. The path preceding this action is part of the stimulus for the action.

The work to be proposed in this experiment is based on the assumption that there is at least one "minimum path" for any modification to a programming system. A minimum path is defined as the one that produces the least number of actions to accomplish the modification. Therefore, an analysis is proposed of the differences in actual paths taken by maintenance programmers and minimum paths in order to identify the factors that cause a programmer to "stray" from the minimum path.

#### 5.4.2 Procedure

Several programming systems will be derived or selected from ones already existing. These systems will represent several types of programs; e.g., data manipulation, calculation, utility, or operating system. For each system, modifications will be specified. The systems and modifications will be simple enough to permit the execution of the following tasks.

##### Task 1: Determine Minimum Paths

Modifications will be performed by the project staff. Then the modifications and the systems will be analyzed to determine at least one minimum path for each.

##### Task 2: Observe Paths Followed by Programmer(s)

Next, an experienced programmer (who is unfamiliar with the preceding work) will make the modifications. During this phase a record will be kept of the path and actions that he takes. The record will also include documentation of verbalized reasons for taking individual path steps and actions.

##### Task 3: Analyze Observed Path Differences

The differences between the minimum paths and the ones taken by the programmer will be analyzed in an attempt to determine the factors that caused him to deviate from the minimum path.

Attempts will be made to relate these factors to differences in programming systems, documentation techniques, etc.

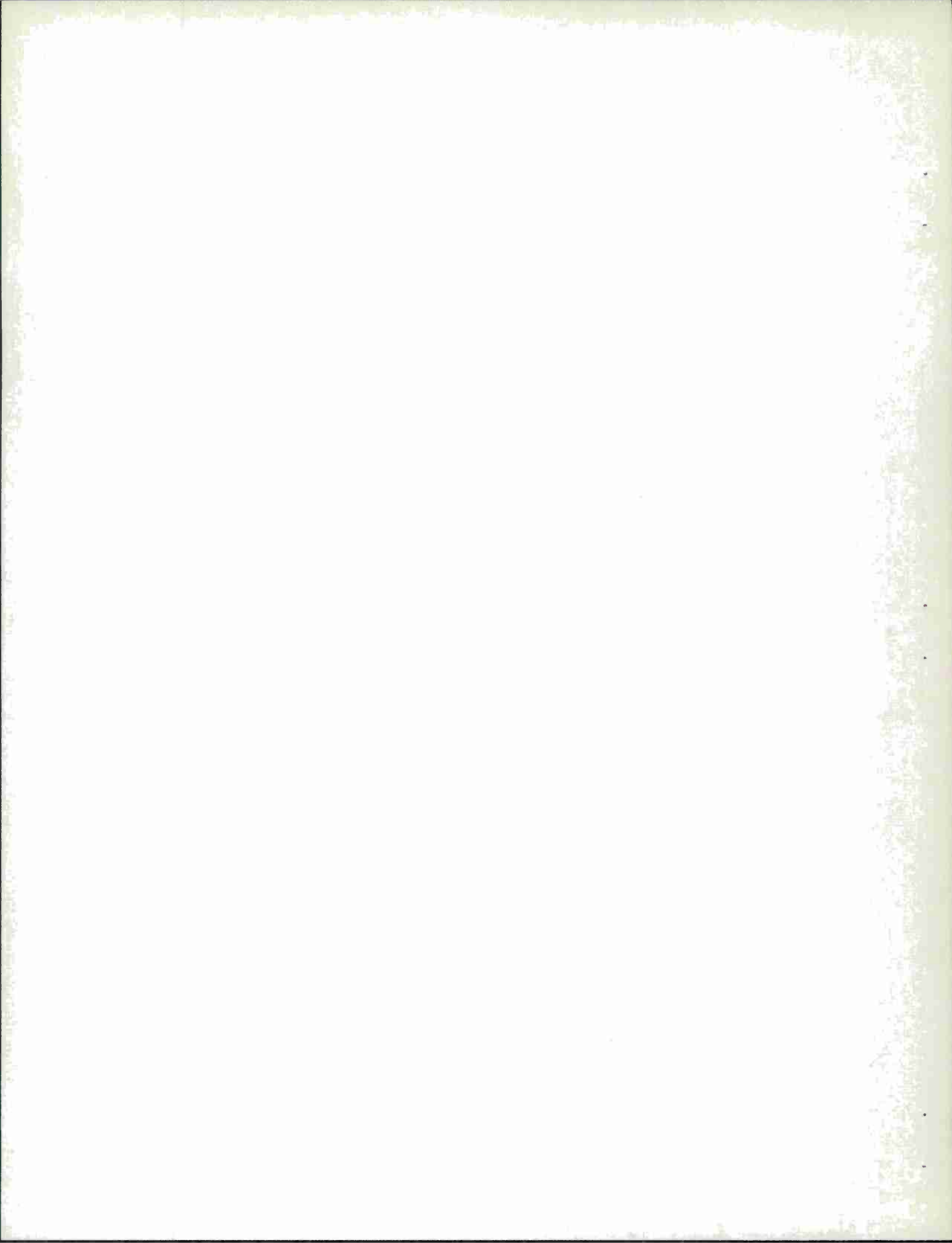


At least three analytic results are expected:

- 1) A preliminary classification of "wrong turns", to produce seriousness-of-effect profiles.
- 2) The generation of ideas and tools for improving the "path-finding" capabilities of maintenance programmers.
- 3) Generalization and documentation of the techniques employed in the analysis so they could be used to analyze proposed solutions for their path-finding effectiveness.

#### 5.4.3 Manpower

The estimated effort for the path analysis feasibility study is 4 man-months.



PART III  
REPORT OF EXPERIMENTAL  
STUDIES



## SECTION 6 - OVERVIEW OF EXPERIMENTAL RESULTS

Many reports begin with a summary. This is not a summary. It is an overview of the experimental phase of the Computer Aided Software Maintenance Study. The principal (or the most interesting) findings appear in this overview. Evaluations of the results should consider the fact that these were pilot studies, representing relatively small efforts.

The authors also present some opinions and general recommendations about future studies. (Later sections include more specific recommendations.)

### 6.1. Principal Findings

#### 6.1.1. Conceptual Groupings

Preliminary experiments, focused on the groupings of code which programmers actually use in explaining a program, yielded findings including these:

1. Some consistent characteristics do appear in programmer-used groupings. These characteristics may be used to improve (or change the nature of) documentation in the future.

2. There is a roughly logarithmic relationship between (a) the length or programmer-used groupings, and (b) the incidence of groups of those lengths.

Such a relationship would appear if the explainers (a) described the programs, or portions of them, at three different levels of abstraction/detail; and (b) devoted about the same degree of attention to each level of discourse; and (c) when moving from one level to another, changed the size of the groupings they used by approximately a factor of four.

However, the data do not prove that the explainers actually did this.

#### 6.1.2. Path Analysis

Is it feasible to trace and analyze the path which a programmer follows in reaching the point at which he "lays code" to make a modification? One study was aimed at answering this question. (The answer was yes.)



It also produced findings including these:

1. The initial information given to a maintenance programmer is of great importance. His errors tend to compound themselves, and misinformation early in his path can lead him far astray.

2. At any given point on his path, wrong, extraneous, or inappropriate labels on variables or statements or comments are a major factor in his erring from the optimum path.

#### 6.1.3. Metrics for Maintainability

This study sought (a) a more precise definition of "distant referent," (b) a metric for measuring "distance," and (c) similar, helpful concepts and metrics.

1. "Distance" is (surprisingly) correlated with "looking around the neighborhood." That is, programmers show a pronounced tendency to overcome "distance" by studying, in nearby sections of coding, instances of the use of the unexplained programming construct; then they try to guess, by analogy, at the clarifying information.

2. The less successful experimental programmer studied distant referents per se more intensively than the other programmer. The more successful programmer "waded in at some reasonable point and sought to learn what the data meant by operation necessity."

3. A major aspect of distance was found to be strange or unfamiliar modes of program control, e.g., recursion.

#### 6.1.4. Methodological Conclusions

An important goal of this phase was the evaluation of methodologies for clarifying the problems faced by maintenance programmers. The conclusions included:

1. The methodology finally followed in the Metrics for Maintainability study (which followed a period of planning and re-planning) was not as appropriate to the goals of the study as it should have been. Recommendations for improving the methodology are presented in Section 8.

2. It is possible to pre-determine a "minimum" path which a maintenance programmer should take. It is possible to compare, with that predetermined path, the path another programmer takes. Such comparisons yield useful information pertaining to fundamental factors inhibiting the effectiveness of maintenance programmers.

3. More understandable groupings of code and documentation can be promoted by the Conceptual Groupings type of study. However, more of the data from such studies should be recorded automatically.

#### 6.2. Overlap in Findings

Three areas were examined during this phase. As knowledge of these areas expanded, the areas began to overlap. In the work plan, no time was specifically set aside for the study of such overlap; so it has received little attention. But its reality is illustrated by the following examples:

1. Results of the Conceptual Groupings pilot study hint at the existence of paths at different levels which would affect the results of Path Analysis studies.

2. Specific coding practices (notably wrong or inappropriate labels on variables or statements) were found, during the Path Analysis study, to lead the programmer astray. Such findings complement the goal of the Metrics for Maintainability study, which was largely aimed at identifying specific features which handicap maintainability and which would be measured.

3. A suggestion which grew out of the Metrics for Maintainability study was this: "When you run into a 'distant referent,' it would be neat to have the compiler generate a statement number, to tell you where it is." If compilers did this, they would greatly influence the Path Analysis results.

4. In the Path Analysis study, the experimental programmer obviously attempted to view documentation in a selective way so attention was directed toward the information that was most pertinent to the task at hand. In this attempt, the programmer was creating conceptual groups from the documentation. Details on this process would help (or might come from) Conceptual Groupings studies.

### 6.3. Discussion and Recommendations

#### 6.3.1. Discussion

Maintenance programming is probably analogous to many other human activities. One of these may be the exploration of a mineral deposit. To wit:

A geologist has an opinion of the location of an underground vein. To test his opinion, a thin hole is drilled (at a cost of perhaps 30 cents per centimeter) to a little below what he thinks is the proper depth (perhaps 200 meters). A programmer holds an opinion about the identity of routines which may require modification. To test his opinion, he goes to those routines and traces their functioning (at a total cost to his employer of perhaps \$10 an hour) until he thinks he has proved or disproved his opinion.

In geology, it becomes necessary to evaluate the deposit more adequately. A new hole must be drilled. Because of its cost, it should return as much information as possible. In programming, further "patching" or testing is usually necessary. Because of its cost, the programmer's additional study and tests should return as much information as possible.

Here the analogy breaks down. In geology, a strategy of drilling rapidly develops. Degrees of confidence from samples are carefully matched against the costs of each sample, and against the cost of erroneously rejecting the geologist's hypothesis if it is really true, or of erroneously accepting it if it is really false. Finally, the work of exploration is deliberately planned so that test shafts can serve not only the purpose of testing, but can be expanded for production later. In maintenance programming, there is often no comparable, explicit strategy.

Because of the dearth of such explicit strategies, it is not possible to say how much would be saved by their use. The opinion of the authors, however, is that savings would be manifold.

The overlap in findings, between the three separate studies, also suggests that the time may be ripe for the use of broader strategies in maintenance programming.

(The reasons for the lack of strategies in maintenance programming may include these: The total cost of each test is usually lower in programming, and the need for a strategy of exploration is less obvious. The computer aspect of the cost may become less obvious to the programmer as time-sharing terminals become more common. Moreover, the programmer does not have to pay his salary for his exploration time; instead, he receives pay. Finally, management may fail to view a software system as a capital asset to be exploited; the system may be harder to visualize than a minerals deposit would be.)

#### 6.3.2. Recommendations

With this overview of the experimental results, the following recommendations are made:

1. Studies like Conceptual Groupings and Path Analysis should be oriented (in future overviews) toward:

i) developing usable strategies for maintenance programmers; and

ii) developing forms of programming and documentation which can be maintained with simple strategies.

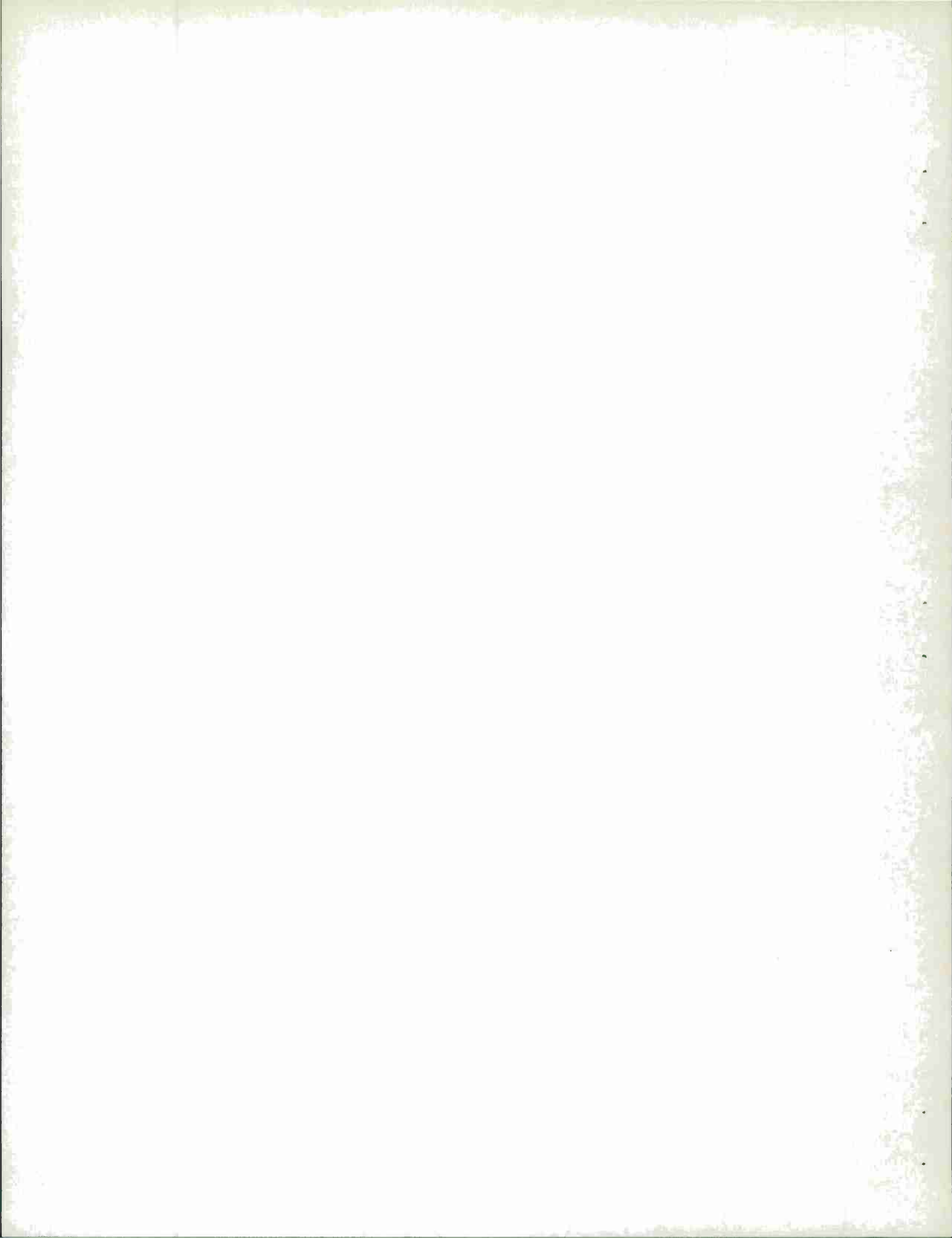
2. Studies like Metrics for Maintainability should be oriented toward:

i) making quantitative data available to the strategists; and

ii) making better tactics and tools available to the programmers.

3. The strategies and tactics should include, among their explicit goals, the minimizing of the handicaps to programmer efficiency caused by the fundamental factors identified earlier.







## SECTION 7 - PATH ANALYSIS FEASIBILITY STUDY

### 7.1 Introduction

Among the previous recommendations was a proposal for an experiment on "Path Analysis". However, it was also recommended that, prior to the formal experiment, a short study be conducted to determine the feasibility of the larger experiment. This section reports the results of that preliminary feasibility study.

The material presented includes the report of the results, of which this introduction is a part, and a set of appendices, including the materials that formed the raw data of the experiment, from which the conclusions in this report were drawn.

The recommendation, based on the experiments, is that the more substantial path analysis experiment and study be undertaken. The remainder of this report details the considerations leading to that recommendation.

### 7.2 Objectives of the Experiment

The central objective, of course, was to determine the feasibility of a research project centered about path analysis. This larger objective can be reduced to several essential components. To determine feasibility, tests were conducted which were microcosms of what was anticipated to be the components of the larger project. These were:

7.2.1 Determination of a "Minimum Path" through a particular program in order to accomplish particular modifications to that program.

7.2.2 Observation, cataloguing and study of a second programmer's deviations from the minimum path to detect obstacles, test the feasibility of the procedure, and determine the facility with which the remedies to obstacles could be detected.

7.2.3 Based on observation, evaluation of the programmer's reference sources and material to determine the traceability of the path through such materials.

7.2.4 Observation, recording and evaluation of the programmer's responses to a questionnaire to determine the feasibility of identifying path-changing stimuli.

### 7.3 Procedure

The procedure employed in the study centered upon four modifications to a particular program made by two different programmers. The steps followed were as follows:

7.3.1 From a group of eight programs an IBM 1130 course scheduling program was selected. It was selected because the documentation was fairly complete and fairly well-organized, the number of statements in the total program was within the range desired (i.e., about 100 statements), and the language, FORTRAN II, was higher order and well-known.

7.3.2 An experienced programmer from the project staff specified four modifications to the program. Then he performed the modifications.

7.3.3 While performing the modifications the staff programmer kept track of his path through the program and the time that it took him to make the modifications. He also recorded various other impressions that seemed pertinent to the objectives of the study. The staff programmer's minimum paths are described in Appendix X. Prior to generating those paths, he analyzed the documentation and prepared supplementary flowcharts.

7.3.4 Next, an experienced programmer who had no previous knowledge of the program was given written specifications of the modifications to be performed. She then made the

modifications (or attempted to do so). While attempting the modifications she followed the instructions in Appendix VIII, kept track of her path through the program and answered various predetermined questions about her efforts. Her recorded responses while making modifications are transcribed in Appendix XI.

7.3.5 The results of the two programmer's attempts at modifications were compared and analyzed.

CIRAD's original procedural plans, and expected results, are reproduced in Appendix VII.

#### 7.4 Results

The following results correlate with the objectives of the study. They are intended to report how the objectives were met, and to cite the problems encountered in meeting those objectives.

##### 7.4.1 Determination of minimum path

The human being's capacity for handling information and making decisions is limited. This capacity varies from one individual to another. See, for example, David (1956). Given such individual differences it is concluded that there is no unique minimum path for all programmers through a particular set of program documentation. There is instead, it is hypothesized, a set of minimum paths. The experiment supported this hypothesis: For each particular modification, each of the three project members who participated in determining the minimum path would have, in general, derived different paths (if each had completely worked the problem individually). However, the technique employed to derive the consensual minimum path seems to be one that could be expanded into a consistent and nearly replicable technique.

The technique followed was this: One member of the project staff questioned an experienced programmer who made the initial modifications about the validity of what he said to be the minimum path. The third project member observed this questioning and asked clarifying questions and



attempted to objectively understand the information that was passing between the two other project members. The criteria of this technique were clarity of communication, adequacy of the path steps to impart the necessary linked information, and detection of extraneous or erroneous path steps. In at least one case, the second programmer (in modification 2) derived a path to the modification that was shorter than the predetermined minimum path. This shorter path was detected and advocated by a project member during the derivation of the predetermined minimum paths. However, the consensus was that the path lengths in question should be retained for clarity.

This incident raises a question of great interest; are there certain types of information in a program that cause or can stimulate quantum jumps in the programmer's thinking? If these types of information, and their structures, can be detected, is it then possible that program documentors can take advantage of such information and structures, and thereby remove factors impeding the effectiveness of maintenance programmers?

#### 7.4.2 Minimum path deviations

There are at least two general conclusions that may be drawn from the observations; (1) the more complex the modification, the more deviant is the programmer's path from the minimum path. The probable cause of this is that for complex modifications there are more choices per path to completion of the modification effort. In the more simple modifications, a limited number of paths make it more probable that two different programmers will take nearly the same path; and (2) wrong steps on a path in a complex modification have a compound effect. In modification 3, the second (experimental) programmer failed to complete the modification. At the outset her deviations from the minimum path became wider with each step she took. Finally, the project supervisor terminated the attempt, at which point the second programmer was on an entirely wrong path for successful completion of the modification. This indicates the importance of the initial information at which the programmer has to look.

Comparative psychology (e.g., Smith, 1966) emphasizes the fact that learning is determined by the nature of the individual as well as by the design of the learning situation. In this experiment the nature of the learning individual (i.e., the second programmer) was not taken into account at all. The design of the learning situation was controlled only by the design of the procedures and the experiment and the selection of the program to be modified. It is expected that in the larger study, that the nature of the individual behavior patterns can be factored out by the selection of a large enough subject population. Control of the learning situation should be improved by the improvement of the design of the questionnaire and better training of the subject programmers.

In summary, the reasons detected for straying from the optimum path are:

- a) wrong or inappropriate labels on variables and statements.
- b) unclear or ambiguous narrative portions of the documentation.
- c) poor organization of the documentation material.
- d) overly complicated coding or documentation -- non-straightforward.
- e) unfamiliarity of the programmer with the language, application, or programming techniques employed.
- f) ambiguous modification specifications.

#### 7.4.3 The program reference material

The reference material for the program was well organized but, within the context of that organization, poorly presented and ambiguous. This ambiguity was



particularly important in the second programmer's attempts at modification 3. The program material contributed to her continuing down a wrong path for a considerable length of time by not conveying the information that would have corrected the path deviation. It was interesting to note that the first programmer made a detailed flowchart of his own that explained to himself the logic of the program in order to supplement the poorly presented documentation. The difference in times of performance between the first and second programmer is largely the result of this documentation which the second programmer did not have available.

The second programmer's responses to the questionnaire showed rather plainly the process of over-filtering extraneous information, and attempting to view documentation in a selective way so that attention was directed towards information that was most pertinent to the task at hand. (See, for example, Broadbent, 1958.) This filtering mechanism, that rejects unimportant data, seems to have been clogged by the poor documentation of this program. One possible avenue of research that is suggested by this is testing the rejection of unimportant data and the acceptance of relevant data versus different documentation forms and types in order to evaluate each type's best adaptation to this filtering process.

Another way of looking at this phenomena is through the so-called principle of economy of memory (Biggs, 1968). This principle is said to describe the process of concentrating attention to those items of information that are highest in satisfying a high priority learning requirement. An interesting question that might be asked: How long is information retained that is learned from various types of documentation?

#### 7.4.4 Programmer's response to the questionnaire

The questionnaire employed in this study was designed intuitively. It is recognized that the application of an instrument must not be based on subjective confidence that the instrument works in practical situations (Magunsson, 1959, pgs 110-118). However, since the intent of this study was to determine feasibility, it was felt that the expense of a full test of the instrument was not warranted if feasibility could be determined using an instrument that was designed intuitively.

The questionnaire was constructed to yield clues as to the stimuli that caused the programmer to seek information. It was also designed to yield the type of information that the programmer was seeking as she went from one "path" element to another. The responses to the questionnaire, particularly for modification 4, indicate a number of reasons or stimuli for the programmer seeking further knowledge elements (or, as we have called them, "paths"). These reasons could be summarized as follows:

- a) Reinforcement. Looking for confirmation of a suspected knowledge fragment in the documentation.
- b) Continuity. Continuing to read sequentially to get "all" information on a subject.
- c) Searching for an answer to a specific question.
- d) Testing a contemplated or executed change.
- e) Trying to resolve an apparent ambiguity.
- f) Seeking more explanations of a confusing aspect.
- g) Attempting to eliminate extraneous information.

Some general limitations of questionnaires, as listed by Telford and Saurey (1968), may have been relevant here. They include:

- a) Lack of frankness and truthfulness, not taking the questionnaire seriously.

The programmer selected has been used in other experiments on the project and had responded to these experiments using a questionnaire. She had a full understanding of the intent of the project. The quality subjectively, of her responses, is one of sincerity and seriousness.

- b) A select group responding.

Obviously, since we only had one respondent, the group was highly select. However, the purposes of this study were more oriented toward determining feasibility than in average group characteristics.

- c) Difficulty in telling how form and order of questions influence responses.

This is one of the areas that in the larger study, assuming that it is undertaken, should address. It should make preliminary tests of various forms of the questionnaire to design an instrument that will be as bias-free as possible. In this particular experiment, the authors did not detect any bias caused by the form and order of the questions.

In summary, the questionnaire seemed to be an adequate instrument to determine if paths, stimuli, and responses could be traced through a set of program documentation.

## 7.5 Recommendations

As noted in the introduction to this report, it is recommended that a full path analysis project be undertaken. The reasons for this recommendation, based on the path analysis feasibility study, may be summarized as follows: The study indicated that:

- a) The concept of "path" in tracing a programmer's progress through programming material is a useful one in delineating the sequence of actions that the programmer takes.
- b) A predetermined "minimum" path can be compared against a path another programmer



takes and that such comparison can yield useful information pertaining to fundamental factors inhibiting the effectiveness of maintenance programmers.

- c) It is probable that the results of such a study could yield specific recommendations that would, if implemented, contribute to the effectiveness of maintenance programmers.

Beyond this, analysis on material like that in Appendix XI (the staff programmer's verbalizations as she worked) could be used for an additional study: How do a programmer's analyses of a program compare with those implicit in the documentation? Can the structure of documentation be made more compatible with programmers' thinking? A study of such questions, while different from path analysis, might be valuable.

The remainder of this section is devoted to an outline of the recommended path analysis project.

#### 7.5.1 Objectives

The objectives of the project will be to (1) derive a well-defined method for describing programmer paths through program material, particularly in terms of objective descriptions of path elements and responses; and (2) to apply this methodology to understand further the causes of programmers' aberrations that inhibit the programmers' effectiveness; and to recommend remedies.

#### 7.5.2 Procedures

Several programs will be selected as experimental devices. Simultaneous with this step will be the performance of a literature search and survey, to update and amplify the body of reference material that has been collected during the CASMS project. Next, an experiment will be designed that will include questionnaires, selection of the subjects, outline of the experimental steps, and analysis of the objectives versus the experimental procedures. Then a test of the experimental design will be conducted. This test will be done by the three members of the project staff as experimental subjects in order to detect deficiencies in the design, and to improve the design. After this is done, there will be a definition of the path elements and response

characteristics which will be used in the analysis of path deviations. Next, the path element design will be tested "dry-running" parts of the experiment, using the path element definitions to determine their adequacy and to improve them. Then the experiments will be conducted, using five subject programmers. Finally, the results of the experiments will be analyzed, and a final report created.

### 7.5.3 Results expected

It is expected that the analysis of the experiments would give the following results:

- 1) The procedure would produce well-defined techniques for applying path analysis to the assessment of the efficacy of various documentation and coding techniques.
- 2) Studies of deviations from optimum paths would yield further delineations of obstacles that impede the effectiveness of maintenance programmers.
- 3) As a result of the analysis, recommendations would be made for specific remedies for obstacles impeding the effectiveness of maintenance programmers.
- 4) Recommendations would be made for methods for grouping and organizing programmer reference material.
- 5) Finally, it is expected that the raw data generated by the experiment would form the basis for further analyses that might lead to other relevant research efforts.



#### 7.5.4

#### Possible applications of results

- 1) It is expected that a technique would be recommended for defining paths through programs, and that the technique could be applied to the evaluation of various methods for structuring programming material.
- 2) Additional specific recommendations would be made for improving the effectiveness of maintenance programmers.
- 3) The technique of path element description might be applied to a documentation and/or debugging aid for maintenance programmers. This application might result in a generalized method of segmenting documentation and material pertaining to programs into more meaningful sequences of information.



## SECTION 8 - METRICS FOR MAINTAINABILITY PILOT STUDY

### 8.1 Introduction

The Metrics for Maintainability Study was one of the areas delineated in Section 5 of this report. as meriting attention during this phase. The motivation underlying this study as there expressed was:

"It would be good to be able to measure the rate and ease with which relevance testing is done; it would be good to have metrics which could be applied to programs to yield such measurements."

The aim of this study was, therefore, to create "the definitions and measures needed for the quantification of rate and ease of relevance testing."

The material presented in this report includes the report of the results of this study and a set of appendices comprising the materials used as inputs to the study and the semi-raw output data derived therefrom. Several findings were surprising to the investigators, i.e., they were not expected. These results and appendices underly the conclusions which are reported.

Finally, recommendations are presented. These recommendations may be summarized as follows:

**Aims:** The aims of this type of study need to be more narrowly defined.

**Procedure:** The methodologies used to conduct this type of study need a great deal more attention than was possible in a pilot study.

### 8.2 Objectives of the Study

The objectives of this study have been stated above in a general form. More specifically, the study had as its aim to look for any phenomena in the programmer's interaction with a program and its documentation susceptible to quantification. One concept vaguely defined at the start was that of "distance" and "distant referent". As stated in Section 5:

"... it seems entirely logical to assume that a maintenance programmer has to do more work (make more relevance tests) when there are more distant referents, and when their distances are greater."

Thus, a subsidiary aim of the study was to arrive at a more precise definition of distance and distant referent. To a certain extent this has been accomplished.

### 8.3 Procedure

#### 8.3.1 Revised Procedure

The previously described procedure of the study was abandoned almost at the outset. Task 2 of the original procedure called for a programming exercise "... aimed at insuring that the concepts being developed ... are applicable to the real world." It was soon realized that the role of the programming effort could only be fulfilled if it were done carefully and well and to do so would take an effort disproportionate to its overall value to the study, especially a pilot study. It was felt that the aim of the study was concept formation and a procedure concentrating on that would be more appropriate.

The revised procedure adopted, consisted of the following tasks:

1. Select an unfamiliar, non-trivial program. The program selected was the AGC Symbolic Assembler written by Mrs. Larkin Veigel, a member of CIRAD's technical staff.

2. Have the program author explain the program simultaneously to two programmers, one a part-time employee and the other the CIRAD staff member in charge of this study.

3. Set a modification goal for the programmers. This modification goal was to be explained as part of Task 2.

4. Develop a checklist/guideline for the programmers to aid them in keeping a magnetic tape log of their work under Task 5. This appears as Appendix XII.

5. Have each programmer study the program in an attempt to install the modification. The attempt does not have to be successful. Each programmer is to maintain a magnetic tape log of his thought processes as he works following the guidelines set up in Task 4. An interpretive, sportscast-style digest of these logs appears in Appendix XIII.

6. Analysis. The analysis of the tape logs appears as part of the results in Section 8.4.

#### 8.3.2 Ad hoc Revisions to Procedure

Numerous defects in this procedure emerged during the study and later upon undertaking the analysis of the tape logs. Those that emerged during the study will be discussed here since ad hoc measures to ameliorate them were instituted; those that emerged later will be discussed in Section 8.4.

The first procedural defect to emerge was that one explanation session was not sufficient. The first session took place on a Friday and by the time it ended, both experimental programmers had agreed that a second session would be desirable after some opportunity over the weekend to digest the program and its ancillary documentation. The second session was held early in the following week. The second session was taped; the first was not.

The second defect in procedure which emerged immediately was that the staff member programmer found it impossible to be his natural self in the context of programming and recording his thoughts on tape knowing that he would later be responsible for analyzing this self-generated data. It was decided, therefore, to bring another programmer into the study. The program was explained to the new programmer by the staff member programmer.

A third procedural deficiency appeared after listening to the first tape produced by one of the experimental programmers (the part-time employee). Since she was still working on the program, it was felt that there was time to correct it. It had been assumed that this programmer's experience together with some instructions during the explanation sessions would suffice to give her enough knowledge about the language in which the program was written (SPL/Mark IV for the CDC 6600) to enable her to fill in the gaps with little effort. This assumption was incorrect and therefore, she was given a two-hour language tutorial before she proceeded any further. The parts of the tape log digest for this programmer before and after the tutorial are indicated in Appendix XIII.

#### 8.4 Results

The results reported here are derived almost entirely from reflection upon the contents of the tape logs



kept by the two experimental programmers. The results fall into two major categories.

The first category encompasses results which reflect upon the aptness and efficacy of the study procedures themselves. It is entirely fitting that the results of a pilot study should be at least partially concerned with methodology. The second category encompasses results bearing on the aims per se of the study (Tape log quantitative/qualitative results).

#### 8.4.1 Study Procedure Results

The tape log digests appearing in Appendix XIII. make it plain that better procedures for this sort of study can be devised. Although we were not seeking or expecting success in making the modification, nevertheless, it was felt that more information could have been derived from such a study by more careful attention to the procedures used. In particular:

1. Selection of the program(s). The program(s) selected for such a study should be relatively well-documented programs that are or have been in actual use. The program used here was a developmental program which was about to be abandoned in favor of a radically different version and which was poorly documented. Ancillary documentation was supplied with it. This consisted of (a) an inaccurate set of BNF formulas for the syntax of the assembly language being handled by the program (an assembler); (b) a relatively accurate lexical scan decision table; (c) a set of pages reproduced from the second chapter of Cocke and Schwartz's book (1970) on writing compilers said by the program author to be "the cookbook recipe" from which she had concocted her program.

2. Explanation of the program. It appears to be a mistake to have the author of a program explain a program to anyone. The author is wrapped up in his private little world of agonies gone through when the program was being written and checked out and is therefore liable to concentrate on the details of these and not necessarily on

The things the novice needs to know. This becomes apparent in listening to the tapes of the second explanation session. In the program author's own words:

"I've never sat down and thought it out as clearly as I should have to be able to explain it to someone else."

A more rational procedure for explaining a program to experimental programmers would seem to be: Have the study investigators digest and learn the program. Having gone through the problems connected with a particular program, they are in a good position to impart (or withhold) whatever information is necessary in the context of the study.

3. Selection of programmers. As noted previously, a person responsible for analyzing data should not also generate it. Furthermore, unless the study specifically bears on how programmers learn unfamiliar programming languages, the experimental programmers should be familiar with the languages in which the programs they deal with are written.

4. Specification of modifications. The modification should be specified in written form with as much or as little detail as befits the objective of the study, but written. In this study, the modification was imparted orally. As a matter of (tape) record, the program author practically gave away the whole show at the second explanation session, but the experimental programmer (the part-time employee) failed to take notes and forgot what had been said when she got to work.

5. Recording results. Even if the experimental programmer is not later going to be responsible for analyzing what he has said, knowing that later someone will listen to what has been said engenders a certain amount of self-consciousness and having to verbalize one's thoughts slows down the rate of thinking or causes utterances to seem disoriented as the mind races ahead of the tongue. One becomes aware that he is filling up the tape with gaps of silence and "uhms". Finally, the impending end of the tape or cassette is anticipated nervously.

What is probably desirable in this regard is some means of spying on the programmer unobtrusively. Using a CRT terminal system to fetch the things the programmer wishes to look at is one means for doing so. The use

of some combination of video taping and voice actuated tape recording is another means which does not, however, obviate the need to speak aloud.

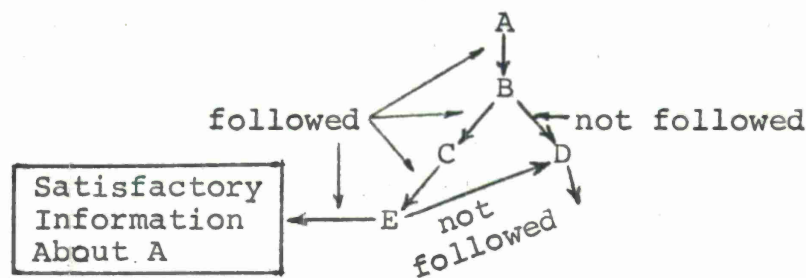
#### 8.4.2 Tape Log: Quantitative Results

We indicate here some feasible directions for deriving quantitative results from the raw data represented by the tape logs. We indicate also why we have not pursued these directions intensively. It is felt that it is sufficient for a pilot study to uncover feasible directions for future exploration; a pilot study is not required to produce a full-blown, polished theory.

It was noted in reviewing the tape logs that the experimental programmers utilized several types of recognizable search maneuvers. These are discussed at greater length in Section 8.4.3. On the basis of a sample of one program, one language and two programmers, it is unreasonable to expect that these search maneuvers exhaust the list of possibilities, but two things stand out. First, there is definite indication that different maneuvers are used to achieve different objectives, and second that some of these search maneuvers exhibit a phenomenon which we shall call "information tree building". It is felt that information tree building contains the seeds of quantification. To justify this feeling we pause to give an example.

Suppose a programmer encounters a program entity about which he wishes to know something, right now, i.e., he will not be content with a cursory, hazy notion about it in hope or expectation that he will later learn more (while looking at something else) or alternatively, suppose he has made a guess about something and wishes to find evidence that supports or disproves his theory. If we call this program entity A, then it may turn out that in order to learn about A he must or would like to find out something about B; he may, of course, be wrong. Nevertheless, B causes him to look at C, and C at D, etc. This sort of pursuit, whatever its origin and despite errors made by the programmer, tends to build up a tree in which the master node is A and the subsidiary nodes are B, C, D, etc. For example:





If we call A a reference and the satisfactory information about A the referent of A, then one measure of the "distance" of this referent from A is the number of intervening nodes. Further, some function of these distances for all the A's in a program could conceivably be at least a partial measure of the maintainability of that program. Examples of trees with measurable length appear in the tape logs.

We have not pursued this direction intensively because the exploratory, pilot nature of this study meant that we were looking for concepts which we would not be prepared to study in depth. Before this avenue can be explored, a great many questions have to be answered and definitions made more precise. For example:

What do you do with an information tree which doesn't seem to terminate or loops? Both experimental programmers encountered this.

What do you do with an information tree in which the information sought is not a sudden jackpot at the end, but is gathered incrementally at different nodes?

How do you measure "satisfactory"? What if the satisfactory information eventually proves unsatisfactory?

How do trees interact with one another?

These matters are discussed again in the section on recommendations (8.5).

#### 8.4.3 Tape Log: Qualitative Results

In general, a programmer's interaction with a program is an exercise in deriving meaning and intent

conveyed in a language which is different or divergent from that in which he thinks or expresses his own thoughts. This in its most protean sense is the origin of all distance, but this philosophical approach does not shed any light on what distance is or how it may be measured; neither do the results appearing in this section. Rather, like the shadow of a man, which neither is nor measures the man, they indicate strongly that distance exists and is very real. These "shadows of distance" take the forms displayed in the following subsections.

#### 8.4.3.1 Distance-Reducing Devices

Distance is shown to be real when programmers take measures to diminish it. Distance reducing devices encountered in the tape logs include the following:

1. Using information in the immediate neighborhood to avoid searches. Specific examples are:

Searching for nearby comments or uses of a variable;

Searching the immediate neighborhood to enumerate and compare the uses of a repeated construct to form a database for analogies. This proved to be a two-edged sword: the programmer who did this to gain information also used analogies to make dubious modifications. This tendency, when encountered, proved to be a surprise to the investigator, but evoked recognition as something he too had done as a programmer.

2. Using program logic to make sense in a situation. An example of this occurs when the second programmer deduces that the function of .MATCH must be to set IND since it isn't set before entry to .MATCH and must have a new value afterwards. This avoids having to find and examine .MATCH.

3. Making use of mnemonics to deduce the function of a variable. Repeated examples of this occur. Often, there are context-dependent clues that aid the process. Thus, for example, in a routine dealing with input/output it is easy to guess that EOFC means "end-of-file character" and RECLN means "record length". Certain mnemonics or subparts become stereotyped in particular contexts.



#### 8.4.3.2 Distance-Increasing Phenomena

The struggle to derive meaning, which we have roughly translated as distance, is protracted by several phenomena. These include:

Physical alignment of the printed source text. The first programmer missed (on her first shot) line 321 of the source text because the text JMP EQ XPROG does not align with the following lines;

Unfamiliarity with programming language. This is evident throughout the first part of the first programmer's log;

Misleading comments, documentation and variable names. Three specific examples of this occurred, the first affected both programmers. (1) The abbreviation RCC, in a comment, standing for recursive control stack, caused both programmers to believe RCC was a variable name and to search for it. (2) The notation "(TEMP)" appearing in the BNF formulas supplied to her caused the second programmer to believe that this was a syntactic literal. It actually was the program author's private notation that the opcodes are read in due to core limitations. (3) The fact that the lexical scan begins with label "JUMP." caused the first programmer to confuse this table with the "jump table" for syntax analysis whose first label is "JMPT.".

#### 8.4.3.3 Search Maneuvers

Distance is shown to be real when programmers adopt means for traversing it. These means are referred to as search maneuvers. They take two major forms.

1. General search maneuvers are undertaken to cover large amounts of source text and are termed "general" because their aim is to spy out the territory, i.e., get an overall feel for what's going on, and not to search for any specific piece of information in particular. It should be noted, however, that general search maneuvers were observed as serendipitous digressions during specific search maneuvers (below). It was noted that these digressions tended to take place during the backtracking involved in specific search maneuvers but were themselves linear. General search maneuvers appear to take two different forms. The first is a simple linear, downward excursion through a portion of the text, as when the second programmer began his examination of the program by going one-by-

one through the data declarations. The second form of general search maneuver takes the form of a tree search with backtracking. An example of this occurs when the second programmer reads a lengthy comment and makes brief sallies into the code each time he encounters a name, but always comes back to continue reading the comment at the point where he broke off.

2. Specific search maneuvers are undertaken to find information about specific program entities. They too may be linear or tree search in type. The latter type, as noted above, has a tendency to promote digressions of a general nature while backtracking. Examples of specific searches include:

- looking for places where specific variable is set in an area of code;
- looking for the meaning of a specific variable by searching for its declaration or for other areas where it is used;
- looking for a label;
- looking for a procedure. Both programmers were aided in this process by noting fairly early that the name of the procedure appeared in card columns 73-80 on the print-out.

#### 8.4.3.4 Recursion

Recursive program control used in a program is a subtle form of distance totally unlike any other form of distance encountered in this study. Any other form of program entity, whether a label, a procedure (non-recursive) or a label is a citizen of Flatland (Rev. Edwin A. Abbott, Flatland: A Romance of Many Dimensions, a classic of mathematical fantasy published circa 1900). The Flatland is the program text and to find anything, the programmer need only move about to somewhere in this two-dimensional space to find out where it is or where program control goes to next. Recursive program control is analogous to the third dimension, especially to programmers who, like the two used

in this study, have seldom if ever encountered recursion before.

One might expect that a simple recursive procedure such as n-factorial would be easily understood by a Flatlander. A more subtle type of recursion wherein procedures A and B call each other but not themselves directly would be more difficult, but in the program used in this study, there is not one single recursive procedure. The recursive control is carried out by storing go-forward and back-up indicators in two LIFO lists, YEST and NOST respectively. The recursive control is effectively hidden from a programmer who doesn't know what to look for. The single method which might have exposed it, detailed "play computer" simulation with sample input data, was avoided by both programmers. The results appearing in the tape logs speak for themselves: pure confusion.

One anticipates that other exotic forms of program control such as asynchronous parallel processes and co-routines would share the third dimension with recursion.

#### 8.4.3.5 Personal Work Habits

The two programmers used in this study exhibited different preferences for distance reducing devices, reacted differently to the distance increasing phenomena and differed in the search maneuvers they used. They also differed in other characteristics which have not been discussed above. The range of observed differences is best exhibited in tabular form as shown in Table 8-1.

This table can best be summarized by observing that the differences observed, the tactics employed are influenced to a great extent by the programming experience/background of the two programmers. The first programmer's background is primarily in business programming using COBOL. This, as noted in the table, inclined her to wade right in and make a modification (wrong, but in the right places) and trust to a computer run to provide her with further information. The second programmer has a degree in mathematics and a varied programming and systems analysis experience in both scientific and business applications. He sought mastery and understanding and never did attempt the modification except that he did recognize the place where the lexical scan was omitting open and close

Personal Work Habit Characteristics	First Programmer	Second Programmer
1. Preference for linear search maneuvers -- particularly scanning variable declarations.	Less	More
2. Thinking and analysis vs action oriented	Action	Thinking
3. Pursued a wrong theory to death	Often	Seldom
4. Selectivity in choosing what to study, what to ignore	Poor	Poor at first but improved
5. Ability to proceed on a path of inquiry on basis of wrong or partial information and make corrections by later discoveries.	Less	More
6. Made effective use of Comments and Literature	Less	Excellent
Program Mnemonics	Less	Excellent
Program Logic	Less	Excellent
7. Made enumerative comparisons to see analogies	Great	Nil
8. Forgot things learned Within Same Session	No	Yes
7 Day Gap Between Sessions	Yes	Not Applicable

Table 8-1: Personal Work Habit Characteristics



parentheses. He recognized the recursive nature of tree building but did not recognize that recursive tree building depends on recursive program control (at least in this program).

## 8.5 Recommendations

This section on recommendations begins with an observation which is more in the nature of a conclusion that arises from the study as a whole and not from any of its smaller parts: The results here seem to indicate that distance is not a sufficient under-pinning for measuring the rate and ease of relevance testing. More relevance tests are required when one does not know what to look at to get the essential information, a fortiori when one does not know what the essential information is. Under such circumstances, the programmer tends to undertake linear searches paying attention to every detail at the same level of importance and has to make more relevance tests to find out what is important and what is not. Under such circumstances he also is inclined toward another fundamental factor, over-confirmation in clues.

Thus, it is recommended that a new and essential step be inserted into the prototype set of procedures implied by Sections 3.3 and 8.4.1. This is: after the program has been explained to the experimental programmers at the level of detail deemed appropriate to the particular experiment, an objective and gradable examination be administered to measure the resident information with which the experimental programmers begin their task. The results of this examination can be used to group the programmers for purposes of comparing and analyzing the results of the experiment. The results of the experiment, likewise could take the form of an objective examination to measure what has been learned during the experiment. This would be in addition to other forms of gathering experimental information such as the use of voice-activated tape recorders, video-tape and CRT terminals, etc.



Other procedural recommendations of a general nature have been given in Section 8.4.1. Additional, specific procedural recommendations follow from recommendations on the aims and scope of future studies in this area. We shall discuss these first and then return to the procedures they imply.

It is our observation that the aims of the present study are too broad, i.e., at too diffuse, abstract and high a level to be sustained in future studies in this area. It is recommended that future studies concentrate on two approaches which are not necessarily incompatible.

The first recommended area of concentration is in the area of feasible directions for quantification indicated in Section 8.4.2. As discussed there, a great deal of thought and sharpening of definitions must take place before launching any effort.

The second recommended area of concentration stems from the observation made in Section 8.4.3: our basic concern is with how a programmer derives meaning and intent from a programming language. It is our opinion that meaningful quantification can only be educed by concentrating on the actual effects (as opposed to the language designer's intended effects) of linguistic minutiae at first and broader linguistic phenomena as more is learned about the effects of minutiae.

The implications for study procedure of this second approach are in the area of the design of statistical experiments. It is evident that the minutiae we have been discussing cannot really be studied in isolation from one another. It is also clear that a few experiments run on a few test programmers will not yield statistically supportable conclusions. Therefore, it is necessary that adequate thought be given to statistical design of analysis of variance/co-variance models and to the replications and controls necessary to produce believable results.

## SECTION 9 - CONCEPTUAL GROUPINGS PILOT STUDY

### 9.1 Introduction

When a programmer is actually working with a listing -- changing it to instal a modification, or explaining it to someone who is going to help him work with it -- he does not just repeat or read the listing, one statement at a time. He says (to himself, or perhaps to someone else): "...Now this gets you into a big old DO-loop; and it does so-and-so; and along the way you go to this other little routine which stores the results; and later you go to a print routine..." Generally, each of the clauses in his thinking covers more than a single programming-language statement.

The moral of this story is that (1) there are some units in which people think about the details of the program, and (2) those units are not limited to statements.

Beyond this moral, another assumption is made: When programmers are studying or explaining a program for their own benefit, they use the explanatory units which best clarify the program. But when they are just meeting the requirements of documenting a program, they choose the units which minimize the work of documentation.

Obviously, it would be valuable to define the units which best clarify the logic of a program. (Reasons why were summarized in Section 5.3. The discovery of such optimally-clarifying units is the general objective which extends beyond this pilot study.

The pilot study was not intended to attain the general objective. (Indeed, in its most general form, the objective may be impossible to attain.) Instead, the pilot study had the following as its major objective: to make sure the study procedure (including materials and personnel) actually produces reasonably valid answers. In

other words, the major objective was to make sure that a possible larger study would be efficient and informative.

A second purpose of the pilot study was to approximate the goal of a possible larger study: To make a tentative definition of "good" explanatory units in objective terms, and then to convert those definitions into techniques for finding the groupings of statements which would be most helpful to a maintenance programmer. In other words, the results represent a small step toward the goal of helping the units of future documentation to be more compatible with those perceived by people. (This compatibility will become increasingly important if a trend toward computer-generated documentation and the integration of documentation and software is maintained.) (The goal will also loom larger if structured programming becomes increasingly popular. This is true because structured programming is "chunk-oriented.")

## 9.2 Method of Study

The Conceptual Groupings pilot study included two sequence of 13 preliminary experiments, each involving the following factors:

### 9.2.1 Personnel

Three people participated in each experiment. One (who, to make the result more realistic, generally did not know the purpose of the experiment, at least in any detail) was the explainer. His job was to explain a program, which he knew well, to a second person: The explainee. (To make the results realistic, both of these people should have some other reason for taking their roles seriously.)

The third person was the observer. He sat close to the first two and took notes.

Twenty-one different people served as explainers or explainees. With one exception, all were full-time employees of computer-oriented companies. The one exception, used in Experiments 1, 2, 3 and 8 (as listed below), was a housewife who had recently completed seven years as a full-time programmer.

The observers were R. Overton, S. Wersan, and D. Becerra.

#### 9.2.2 Materials

Programs to be explained were the experimental materials. The programs were chosen partly on the basis of length and complexity: The experimenters thought each could be explained in about an hour. (In actuality, the sessions seemed to average about an hour and twenty minutes each; but they were not deliberately timed.)

The languages of the programs, together with the name of the observer and some comments, are listed below.

<u>Experiment Number</u>	<u>Program Language</u>	<u>Observer</u>	<u>Comments</u>
1	COBOL	Overton	Experiments 1 through 3 were primarily for preliminary orientation.
2	COBOL	Overton	
3	COBOL	Overton	
4	FORTRAN	Overton	
5	COBOL	Overton	
6	FORTRAN	Overton	Same program as for 4.
7	FORTRAN	Wersan	Same as for 4.
8	COBOL	Wersan	
9	COBOL	Becerra	
10	JOVIAL	Overton	
11	COBOL	Becerra	
12	COBOL	Becerra	Most realistic conditions
13	COBOL	Becerra	
14	COBOL	Becerra	
15	FORTRAN	Becerra	



### 9.2.3 Procedure

As the explainer described his program to the explainee, the former presumably (and "naturally") grouped statements and comments together into the units which he thought were most meaningful. It was the job of the observer to notice these groupings. During the first three experiments, which were most preliminary, he merely observed. During the last ten, he jotted down at least some notes which would enable him to identify the groupings for analysis at a later date. (In some experiments, note-taking had to be very sketchy because of the speed at which the explanation progressed.)

There were no tape or video-tape recordings of the procedure.

### 9.3 Results of the Study

A pilot study serves several purposes. It gives the investigators an opportunity to find any obvious problems which may grow out of the methodology employed in the study. It provides impressions, which may not be statistically justified but which still derive from experience, of some major factors accounting for the results. Finally, it may yield a first approximation of "hard" data which could be applied to some kind of practical design.

Results relating to these purposes are described below.

#### 9.3.1 Methodological Observations

The experience of the pilot study suggests the following points regarding the study methodology:

1. General conclusion:

No obvious problems developed as a result of the methodology; its "face validity" seemed good.

2. Regarding personnel...:

The degree of realism of the explanation/assignment -- and, by implication the degree



of motivation of the programmers -- varied during the study. But the results were reasonably consistent. Thus, a very high degree of motivation may not be necessary to informative results.

3. Regarding procedures....:

The physical arrangement which seemed most productive was that in which the observer was sandwiched between the explainer and explainee, but took notes unobtrusively. When the observer as well as the explainee asked questions, the observer's perception of units may have distorted that of the explainee. Recording of questions by the explainee should have been more adequate.

Failure to time the experimental sessions prevented the collection of what might have been an interesting datum.

4. Regarding materials....:

In spite of obvious differences between languages (e.g., the major divisions in COBOL and FORTRAN), some consistent patterns (cited in Sections 9.3.2 through 9.3.4) seem to transcend languages. Such patterns would seem to merit particular study.

Difficulty of the programs used here was meant to be moderate, and typical of the level of difficulty most frequently encountered in actual work. If more difficult material were used, one might theoretically expect the boundaries between groups to become sharper, at least in terms of there being longer pauses between groups. This phenomenon has been demonstrated in experiments in concept learning (Wells, 1970). But then the results would be less typical, and the explainers and explainees might be less cooperative.

A video tape recorder would have facilitated the data collection. A disappointment in this pilot study was that many of the data, in terms of notes on code sheets etc., were not understandable after the conclusion of an experiment. Only general impressions remained.

Two types of materials cited in Section 5.2 (page 66) proved to be practically unobtainable. One was called "documentary data." An example of coding in structured programming would have been documentary data if the coding could also have been automatically grouped syntactically by a procedure such as that of Mills. Then it would have been possible to see what contrasts there are between the "chunk-oriented" structured programming units and the automatically-produced "documentation units." But such cases were not found.

Second, it had been planned to contrast the units the programmer actually uses in his explanation with the units used in the formal documentation. But the style and the amount of documentation varied so greatly that meaningful comparisons were not possible.

### 9.3.2 Statistical Results

Most of the parameters of programmer-used conceptual groups have not been defined well enough to permit them to be measured precisely. The length or size of groups, however, could be measured -- at least in terms of number of statements per group, and in those experiments in which the notes were sufficiently detailed to permit accurate counts.

This counting was done on the results of experiments 4, 5, 9, 11, 12, 14, and 15. The results are summarized in the following table:

<u>Length of Group</u>	<u>Percentage of Groups</u>
1 statement	14% of all groups
2 statements	12%
3 or 4 statements	19%
5 to 8 statements	21%
9 to 16 statements	7%
more than 16	11%
undefined number	16%
	<u>100% = 291 groups</u>

Thus, clearly-defined groups used by programmers tend to be rather short: 40% are between 3 and 8 statements in length.

The table also indicates that the length of a group is roughly a logarithmic function of the incidence of the group.

Programmer-used groups were also classified (although less precisely) according to the kinds of statements which seemed to define the groups. The classification was done in two steps. First, each of experiments 9 through 15 was reviewed, and correspondences between groupings and statements were noted. Second, these notations were rearranged under different types of statements.

The results indicated that the following types of statements, listed in decreasing order of importance, were most important in defining groups: (1) IF, ELSE, FOR; (2) DO, PERFORM; (3) GO TO, EXIT; (4) WRITE, READ; (5) Comments; (6) TRANSFORM.

The notations under each of these types are copied in Appendix XIV.

### 9.3.3 Subjective Results

The principal lesson learned subjectively from the study relates to the concept of "pay-off." In particular, the programmers' explanations tended to focus on what the programmers seemed to think were pay-off points. That is, explanatory units tended to build toward points at which the program either produced the desired outputs, or stored the results necessary for the desired outputs to be printed later, or overcame barriers to the desired outputs.

For example, a check-writing program had to calculate a deduction; the calculation was treated as an explanatory unit. The program had to cause a number of X's to be printed before a number; the set-up of these X's formed an explanatory unit.

More generally, a simulation program had to update certain parameters; each cyclic updating of each parameter tended to be an explanatory unit. It also had to determine if subsequent events had taken place; this determination was a small unit of explanation.

In what seemed a very common kind of situation, a resource allocation program had to determine certain requirements by calling certain subroutines. Since the calls seemed to be steps toward pay-offs, they tended to be cited in brief explanatory units.



More generally, anything that completed a stage of a project might represent a pay-off. Once he has his "tools" at hand, a worker (or programmer) might think, he has successfully completed one stage of his work ... even though he might not have begun the work per se.

Another observation was also related to the standards for hierarchies for groupings and sub-groupings: Programmers seem to have no fixed standards. Instead, the importance of a new explanatory unit seemed to be relative ... usually relative to the importance of what the programmer had just been talking about.

If a programmer is going to learn a program to a given level of competence, he probably needs to apply the same standards of study throughout the program. Thus, one could argue that a more systematic approach would be beneficial to a new maintenance programmer; the "most important" units explained to him would consistently be the most important, and the next would really be the next most vital, etc.

Finally, the statistical results tended to confirm the impressions that an explainer simply cannot explain too many statements in one unit; a new programmer cannot grasp them.

Returning to the first conclusion, a basic problem remains: It is to translate the subjective recognition of a pay-off into some kind of objective procedure. This might prove to be a fairly complicated pattern-recognition project.

Additional comments relevant to the pattern-recognition problem were made by one of the explainers, Jan Overton, who explained four COBOL programs during the study. After explaining the first three, Mrs. Overton drew the following conclusions from the exercises:

"The criteria for recognizing chunks or groups of a program vary with the nature of the chunk. Sometimes a chunk is obvious, as in the case where a paragraph is one chunk. Sometimes a paragraph will have subgroups each with a paragraph name. For example, a routine to print page headings might be considered a group. The instructions to print each of five lines making up the page heading would then be a subgroup. Another obvious group might be read routines and write routines. A read routine is typically one paragraph consisting of a few statements like the command to read, another command to store the read data, and another to increment a counter to count the number of records read.

"Other groups or chunks are not obvious at all. Often in the real working part of the program there are more numerous and smaller chunks. One statement may constitute a chunk. For example, an IF statement or decision point could be a chunk; or, sometimes, a series of IF statements comprise a group. For example, the statements could organize a program with two input files, and three field control keys. The routine to test the control fields and keep the files in sequence with each other would be one chunk, and would contain subgroups. The test for each possible relationship would be a subgroup.

"Sometimes it is very difficult to recognize a group. The grouping process is indeed subjective and imprecise. In fact, a given programmer, in explaining the same program on more than one occasion, will probably use different groups.

"In general, in the exercise of identifying groups, the more important something is to the program, the smaller the group. That is, the more unique a chunk is to a specific program, the smaller the chunk will be. Also, areas of a program which are complicated would be in small chunks.

"There are some types of groups which are common to many programs. Some are so common that they might be copied exactly from one program to the next. WRITE routines are sometimes copied. An example of groups which are common is the routine to initialize files and work areas before using them.

"One good way to identify the end of one group and/or the beginning of a group is by observing BRANCH commands or GO TO statements. The sending statement, like a GO TO, is often the end of a



chunk. The "sent to" statement is almost always the beginning of a group or chunk. An example of a "sent to" statement is the receiving address resulting from a BRANCH command.

"In the most fundamental sense, a group is anything that performs a certain function. And, most generally, the group is recognized in one of two ways: first, by perceiving the gestalt of the function (especially when one is explaining one's own program); second, by noticing statements which statistically or logically tend to mark the limits of groups (such as GO TO statements)."

As indicated, some code characteristics do seem to correspond to explanatory unit limits. Procedures stemming from these correspondences are the subject of the next section.

#### 9.3.4 "Prescriptionized" Results

A pilot study partially serves the purpose of approximating or making a first estimate of data which can later be put to use in some kind of practical design. This purpose will be served here by actually describing the designs which might ensue from the data.

Since the study deals with computer programs, it seems natural to report the results in the form of computer flowcharts. (The procedures in the charts could, of course, be manually followed.)

The imagined environment for the application of the flowcharts is the following: There is a large software system in which the code is, or contains, the documentation. A maintenance programmer must go to a console to retrieve any portion of the documentation. The portion which he does retrieve is influenced by the implicit groupings within the code. Furthermore, the retrieval system tries to recognize these implicit groupings and present them to the maintenance programmer. How does it accomplish the recognition? This question is answered by the following flowcharts.

Except for the logically trivial Master Program, each flowchart is simply a statement of an objective piece of advice, or a "prescription," for recognizing the demarcations of conceptual groups within the program. (One should not lose sight of the fact that a prescription here is -- as is sometimes the case in medicine -- an objective statement of the implications of an opinion which derives from some case studies and theories.)

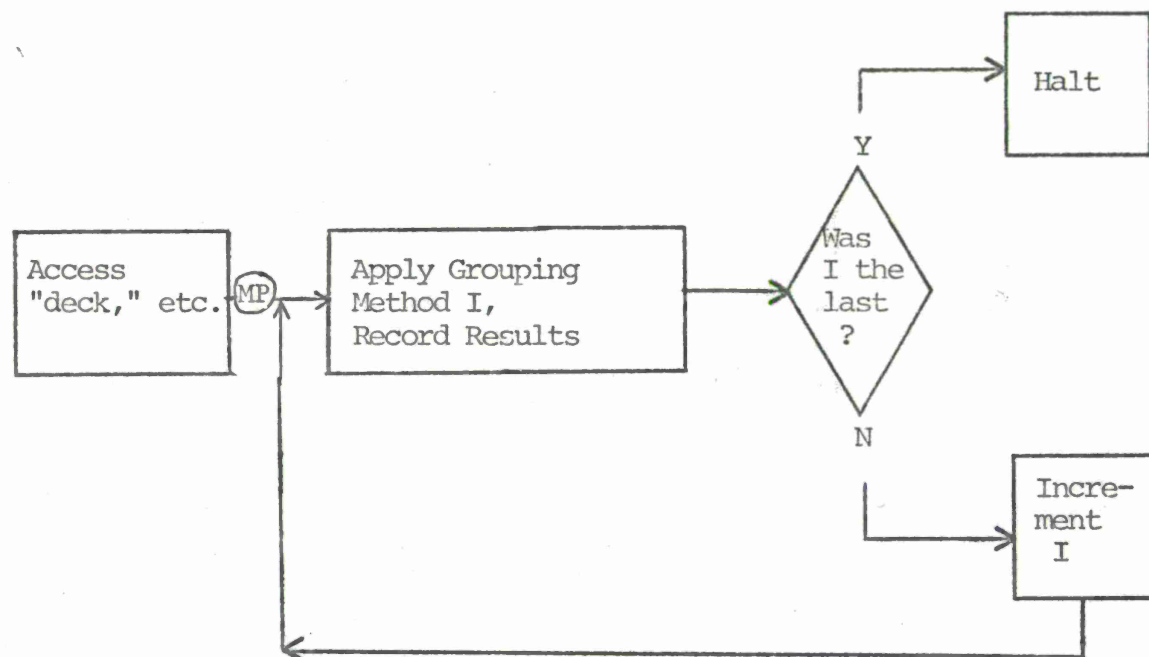
For the combined results to be of greatest value, each "prescription" should be an independently-operating program, making its decisions without reference to the results of the other programs.

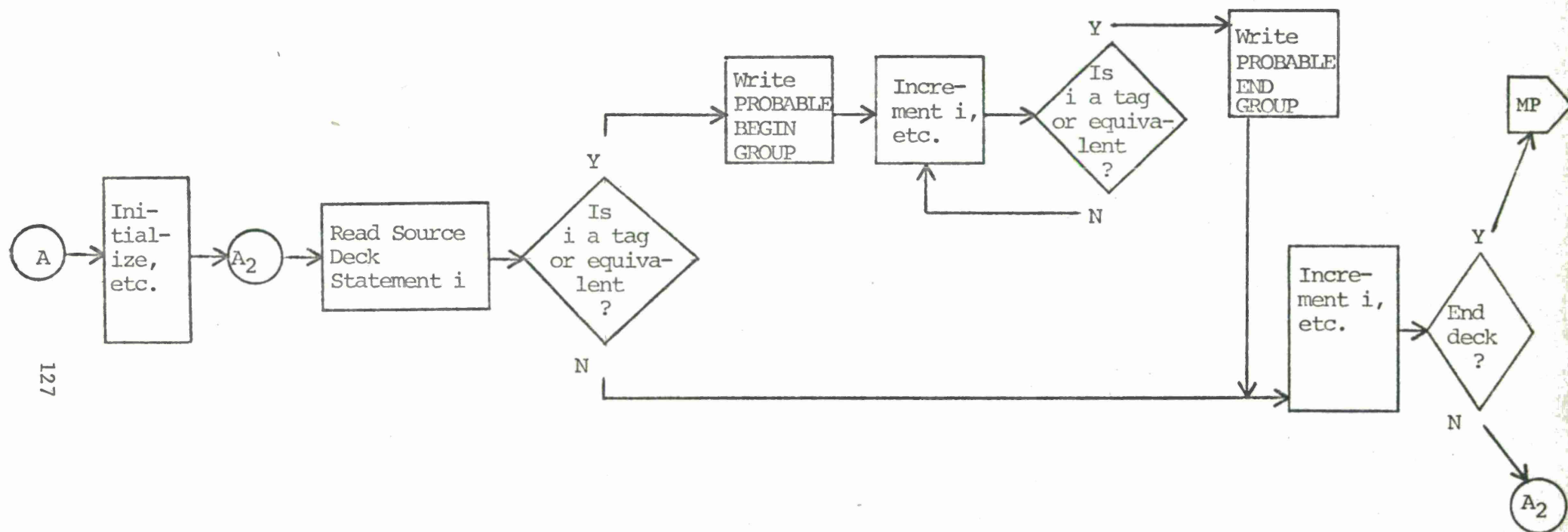
The flowcharts are incomplete in three ways: (1) least importantly, instructions such as "Increment i, etc." are incomplete; (2) certain tests and parameters are defined either incompletely or not at all; and (3) most importantly, perhaps, there are only six "prescription" programs, and more would probably be necessary in a real situation.

Following the flowcharts is an illustration of the output which their logic would produce. The illustration is based on an actual COBOL inventory program being used at Varian Data Machines; the listing has been manually altered to show the additions that would be made by flowcharts like these.

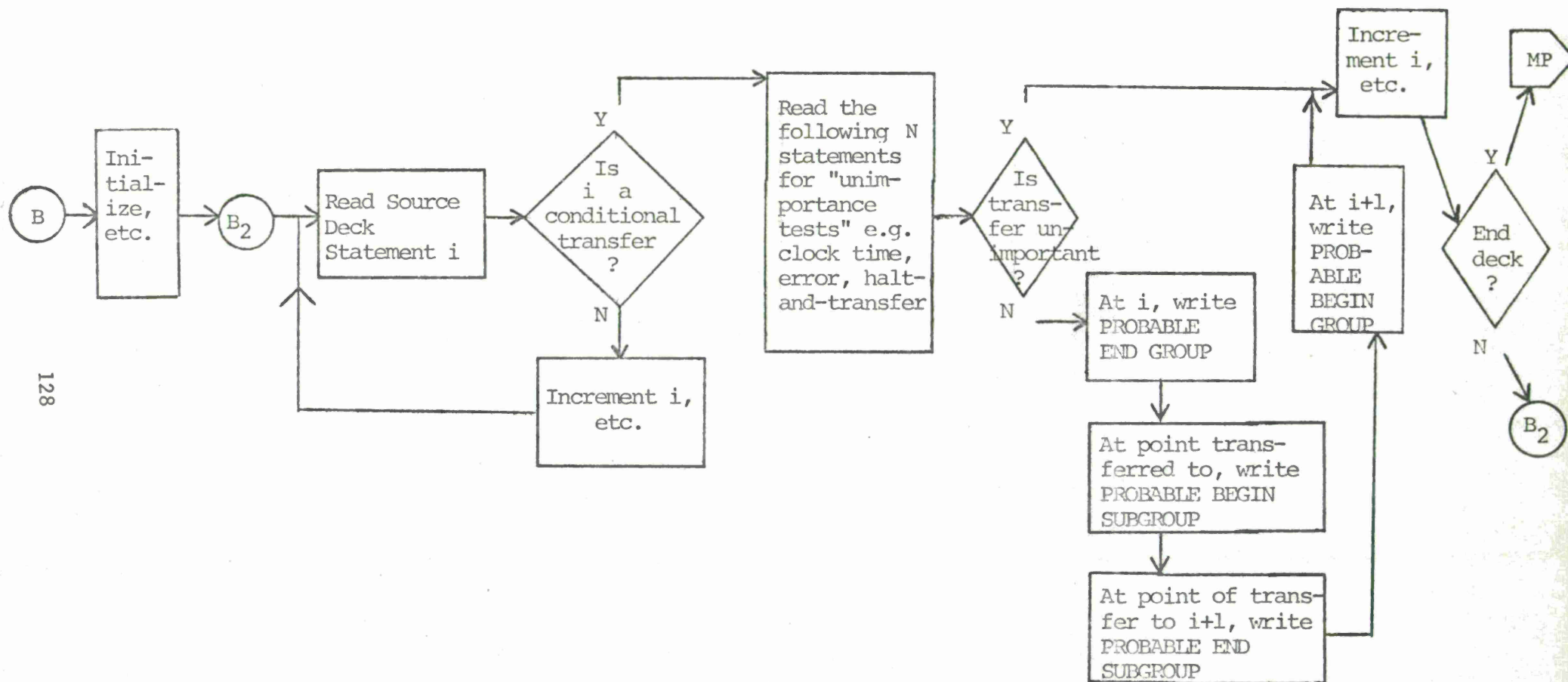
MASTER PROGRAM

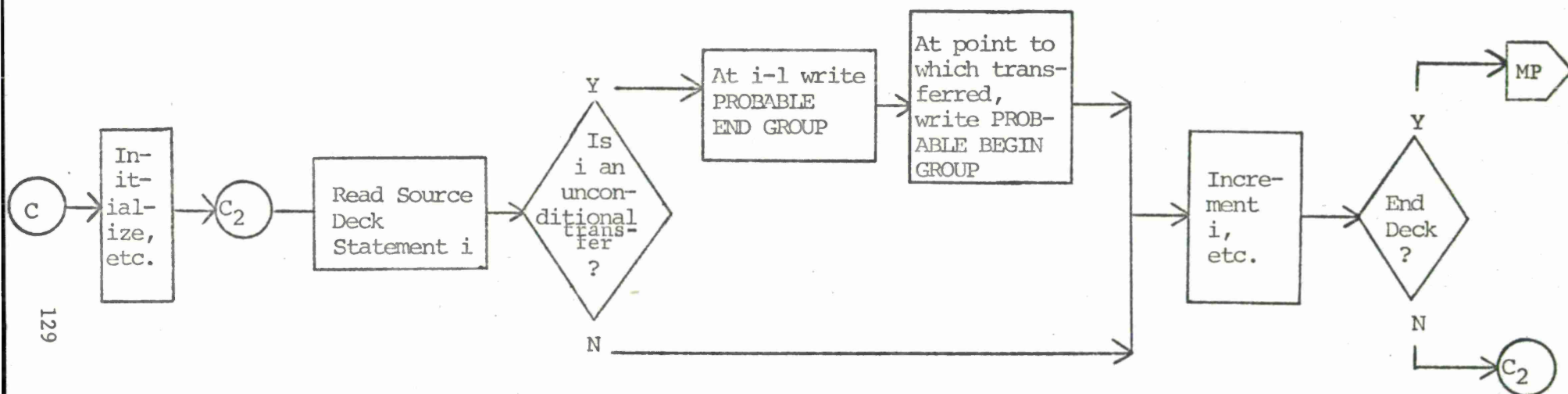
126

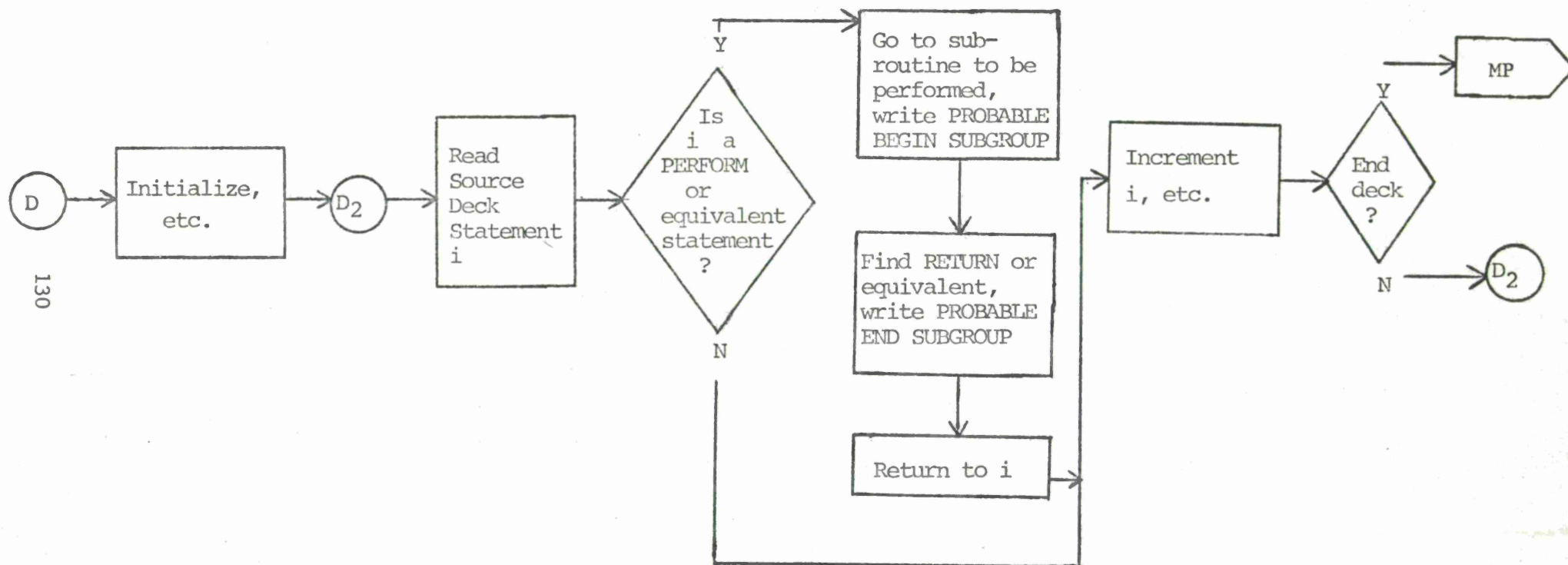


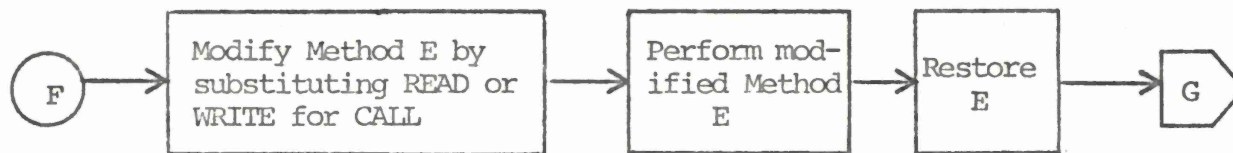
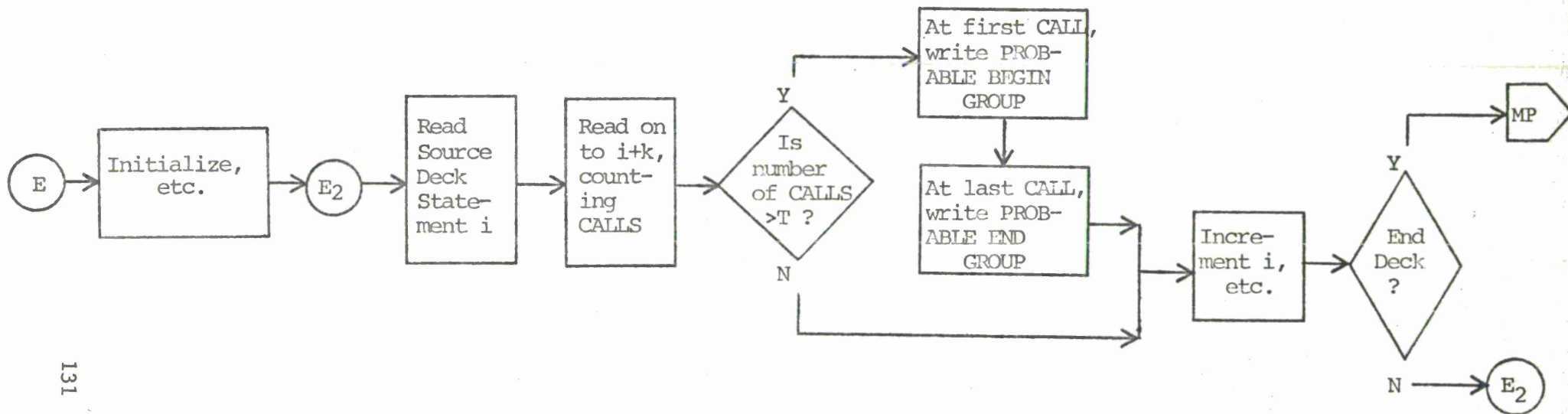














# ILLUSTRATION

OF INSERTIONS IN ACTUAL PROGRAM MADE BY SIMULATED GROUPING PROGRAM

<u>Insertions</u>	<u>Actual (Original) Program*</u>
	.
	.
PROBABLE END GROUP	TOBFSD-END. EXIT.
PROBABLE BEGIN GROUP	TEST-INV-SCHDAT
	IF ITI-TRAN IS EQUAL TO '20'
	ADD ITI-QTY TO A-TPSHIP-QTY
	ADD ITI-AMNT TO A-TPSHIP-AMT
	GO TO TISD-END.
	IF ITI-TRAN IS EQUAL TO '11' AND
	ITI-AMNT NOT NEGATIVE
	COMPUTE ITI-AMNT = ITI-AMNT * -1.
	IF ITI-TRAN IS EQUAL TO '11' AND
	ITI-QTY NOT NEGATIVE
	COMPUTE ITI-QTY = ITI-QTY * -1.
	ADD ITI-QTY TO A-TPFUT-QTY.
	ADD ITI-AMNT TO A-TPFUT-AMT.
PROBABLE END GROUP	TISD-END. EXIT.
PROBABLE BEGIN GROUP	IF NBF-SCH-SHIP IS LESS THAN DC-ENDOP OR
	NBF-SCH-SHIP IS EQUAL TO DC-ENDOP
	ADD NBF-QTY TO A-EOPD-QTY
	ADD NBF-AMNT TO A-EOPD-AMT
PROBABLE END GROUP	GO TO TNBFSC-END.
	ADD NBF-QTY TO A-EOPFUT-QTY
	ADD NBF-QTY TO A-EOPFUT-AMT.
PROBABLE END GROUP (AND)	
PROBABLE BEGIN SUB-GROUP	TNBFSD-END. EXIT.
PROBABLE BEGIN GROUP	ENTER LINKAGE.
	CALL BLD-PRNT USING A,N
	CALL BLD-CLAS USING A,N
PROBABLE END GROUP	CALL BLD-WRIT USING A,N
	ENTER COBOL.
	MOVE ZEROS TO ACCUMULATRS.
	MOVE DTL-LINE TO PRNTLINE.
(Sub-groups labeled elsewhere)	PERFORM WRITE-A-LINE THROUGH WAL-END.
(Sub-groups labeled elsewhere)	PERFORM CLR-DTL-LINE THROUGH CDL-END.
PROBABLE END GROUP	BP-END. EXIT.

# ILLUSTRATION (continued)

## Insertions

## Actual (Original) Program\*

PROBABLE BEGIN GROUP

READ-OBF.

IF INV-EOF-SW IS EQUAL TO 'X' AND  
NBF-EOF-SW IS EQUAL TO 'X' AND  
OBF-EOF-SW IS EQUAL TO 'X'  
GO TO EOJ.

IF OBF-EOF-SW IS EQUAL TO 'X'  
GO TO ROBF-END.

READ OLD-BAL-FWRD INTO OLD-BFIN AT END  
MOVE HIGH-VALUES TO OBF-MODL, OBF-PROD,  
OBF-KEY

PROBABLE END GROUP

MOVE 'X' TO OBF-EOF-SW  
GO TO ROBF-END.

ADD 1 TO IN-COUNT OBF.

IF OBF-MODL IS EQUAL TO '9999999999'  
GO TO READ-OBF.

IF OBF-TYPE IS EQUAL TO '9' GO TO  
READ-OBF.

MOVE OBF-PROD TO OBFK-PRD.

MOVE OBF-MODL TO OBFK-MDL.

IF OBF-PROD IS EQUAL TO 'X'  
PERFORM XOBFB THRU XOBFB-END  
GO TO READ-OBF.

IF OBF-PROD IS EQUAL TO 'Z'  
PERFORM ZOBFB THRU ZOBFB-END  
GO TO READ-OBF.

PROBABLE END GROUP (AND)  
PROBABLE BEGIN GROUP

ROBF-END. EXIT.

PROBABLE BEGIN GROUP

READ-INV.

IF INV-EOF-SW IS EQUAL TO 'X' AND  
NBF-EOF-SW IS EQUAL TO 'X' AND

.  
.  
.

(End of Illustration)

\*Some paragraphing added to original program by simulated grouping program.

## 9.4 Discussion

### 9.4.1 Scaling Up and Down

If a physical thing is simply scaled up or down, new characteristics may emerge. If a man were scaled down drastically, but retained his proportions, he would, for example, lose a greater proportion of his body heat in a cold climate; the ratio of his surface area to his mass would have increased. Now it is possible that the characteristics of "good" explanatory units would also change if it were necessary to scale the units up or down.

Indeed, the pilot study results hint that the characteristics of conceptual groups change with their size. There was a roughly logarithmic relationship between the size of a group and the frequency with which a group of that size was used. This relationship could be explained by the following hypothesis: Large groups were used in explaining a program at one level of discourse, and smaller groups were used in explaining it at the next most detailed level of discourse, and so on; with each change in level, the amount of detail changed by roughly a factor of two. More realistically, explanations may have described programs at three levels of discourse; they may have devoted roughly equal attention to each level; and, when changing levels, they may have changed the size of their experimental groupings by approximately a factor of 4.

On an intuitive and non-quantitative basis, many people have already accepted a hypothesis of this type. It is implicit in structured programming. In the semi-popular literature, a recent article (Vander Noot, 1971) gives advice on testing and argues for three levels: modules, programs, and systems. (The article makes no reference to structured programming, or to Mills and Dijkstra.) The article states that these levels should be carried to documentation. (The article is not cited here for its information content, but to illustrate what is being published.)

Now what happens if one tries to explain a program at only one level of detail? The opinion of the authors is this: If the explanatory units are too small, there may be too many of them, and the explainees may lose track of what is happening. For "understanding" in such a situation, there would have to be a break-down into something like "macro-units" and "micro-units".

In small programs which do only one thing, (such as mathematical programs to perform Fast Fourier Transforms or Kalman Filtering, or a logical program to perform a sort on variable length records), the whole program might be considered to be the macro-unit, and explanation could continue at the micro-unit level.

However, on large systems (such as a real time command and control system involving two computers driving consoles, search and tracking radars and other equipment as well as performing varied computations, or a computer batch processing system handling I/O, time sharing and background processing), it is essential to make the breakdown into more than one level: first, say, in terms of macro-units (which would be a module that drives the tracking radars in the first instance, and a module that reads from the card reader in the second instance), and then in terms of micro-units.

The explanation of macro-units should remain at a general level (what does the module do rather than how). All the macro-units should be explained before any micro-units are treated. This gives the explainee an understanding of the whole job before he has to worry about detail. Too often the explanation of a large system (whether the explanation is verbal or in the form of flow-diagrams) degenerates into details about "FOR" loops and flags, and the explainee is lost in the trees and can't see the forest.

The problem of "macro-units" versus "micro-units" may not be encountered on small programs. The breakdown into "macro-units" becomes mandatory in large systems (such as the two mentioned above), which are, in the vernacular, a "can of worms". These usually have been written by several different people over a period of years; they have dozens of modules or "macro-units" which provide data for one another and share common storage locations. Here, nothing can take the place of some kind of explanation of the macro-units and what they do.



#### 9.4.2 Implications for Languages

It is probably quite obvious that these opinions have implications for programming languages. The implications will not be elaborated here because they are beyond the scope of this study, and because other workers are concerned with them. For example, a recent review of "mini-languages" (Ledgard, 1971) points out that these languages require a programmer to make detailed specifications which result "in a loss of the higher-level geometric notations" (p. 146).

Similar observations can be made about common languages. For example, a conditional transfer in FORTRAN requires one to first grasp the quantity that is being calculated, and then to refer to each of the three places to which transfer may be made (depending on whether the quantity is negative, zero, or positive). On the other hand, a similar function could probably be treated as one conceptual "chunk" in a language which included an IF-THEN-ELSE feature.

It seems clear that higher-level languages at different "levels" will evoke different conceptual groupings. However, the deliberate creation of different languages for different levels of abstraction would be a very major project.

#### 9.5 Recommendations

The following recommendations are felt to be justified by this study:

##### 9.5.1 Continuation

As an aid to the ultimate improvement of the efficiency of maintenance programmers, there should be continued development of techniques for defining meaningful conceptual groupings. Given the improvements recommended below, the present methodology is acceptable for that developmental work.



#### 9.5.2 Languages

In the first stages of further work, studies involving different computer languages should be segregated; parallel efforts should continue with each of two or three significant languages. At the next stage, patterns of grouping which transcend languages should be identified. These will merit special study.

#### 9.5.3 Data Recording

The data recording system should be improved, to permit more thorough observation of the grouping patterns found useful by programmers. This improvement could take either of two forms:

First, the experimental procedure could simply be video-tape-recorded.

Second, the procedure could be conducted at a terminal through which source code or other sequential material could be searched. The programmers would have to instruct the console system to advance or reverse to other material. Their instructions could easily be recorded; this record would supplement the observer's notes.

An advantage of this second improvement is that it would contribute to automatic documentation. As algorithms were developed for automatically identifying groups, the utility of these algorithms could be tested. Later, those which were most useful, and least costly, could be implemented in a practical situation.

#### 9.5.4 Coordination

The next studies should be coordinated with any related projects, especially in three areas:

One area relates to levels of discourse, and this question: At a given level of description, what degree of abstraction (or conversely, what fineness of detail) should be included to make that level homogeneous? (A project in this area may be under way at MITRE.)

Another area relates to Path Analysis. The main work in that area is the tracing of paths and the analysis of the mistakes a programmer makes in reaching a starting point. But paths have to be described in terms of some kind of landmarks. And this study is trying to identify the landmarks.

Also, much of the work of collecting data on paths could be reduced by using techniques like those which are likely to be developed if terminals are used in extensions of this study. (A proposed project in Path Analysis is described in Section 7.5 of this report.)

A third area is systems research as practiced at Carnegie-Mellon University, and as illustrated by Newell et al. (in press).

#### 9.5.5 Practicality

To the maximum extent possible, the next studies should be oriented toward producing results which would lower the cost of program maintenance operations by improving their efficiency. The lowering of costs could take two forms:

First, automatic documentation, if it proved cost-effective, could render the capital investment in existing software systems more valuable, simply by extending their lifetimes by increasing their capacity for modification.

Second, programming standards for new systems could reflect the characteristics which make their units more understandable, and, therefore, which make the systems more modifiable. This would in turn increase their lifetimes and their value.

## REFERENCES

- Baecker, R. M. "Experiments in On-Line Graphical Debugging: The Interrogation of Complex Data Structures." Proceedings, Hawaii Conference on Systems Sciences, pp. 128-129, Jan. 1968.
- Balzer, R. M. "EXDAMS - Extendable Debugging and Monitoring System." Proceedings, Spring Joint Computer Conf., 1969.
- Bernstein, W. A. and Owens, J. T. "Debugging in a Time-Sharing Environment." Proceedings, Fall Joint Computer Conf., pp. 7-14, 1968.
- Biggs, J. B. Information and Human Learning. Sidney, Australia: Cassell Australia L.
- Boehm, B. W., Seven, M. J., and Watson, R. A. "Interactive Problem-Solving - An Experimental Study of 'Lockout' Effects." Proceedings, Spring Joint Computer Conf., AFIPS, pp. 205-210, 1971.
- Broadbent, D. E. Perception and Communication. London: Permagon, 1958.
- Cheatham, T. E. The Recent Evolution of Programming Languages, to be presented at the IFIP Congress 71, Ljubljana, Yugoslavia, August 23-28, 1971.
- Chesley, G. D. and Smith, W. F. "The Hardware-Implemented High Level Machine Language for SYMBOL." Proceedings, Spring Joint Computer Conf., AFIPS, pp. 563-573, 1971.
- Church, A. Introduction to Mathematical Logic. Princeton University Press, 1956.
- Cocke, J. and Schwartz, J. T. Programming Languages and Their Compilers, Preliminary Notes, Courant Institute of Mathematical Sciences, New York University, New York, N. Y., 1970.

- Cohen, J. Chance, Skill, and Luck. Penguin: Baltimore, 1960.
- Cowart, B. E., Rice, R., Lundstrom, S. F. "The Physical Attributes and Testing Aspects of the SYMBOL System." Proceedings, Spring Joint Computer Conf., AFIPS, pp. 589-600, 1971.
- Cumming, R. W. "The Analysis of Skills in Driving." Australian Road Rsch., 1, pp. 4-37, 9, 1964.
- David, R. "The Limits of the Psychological Refractory Period." Quarterly Journal of Experimental Psychology, 8, 24-31, 1956.
- Dijkstra, E. W. "The Structure of 'THE' Multiprogramming System." Comm. ACM, 11, pp. 341-346, 1968.
- Dijkstra, E. W. Notes on Structured Programming, Technische Hogeschool Eindhoven, 1969.
- Edwards, W. "Subjective Probabilities Inferred from Decisions." Psychol. Rev., 69, pp. 109-135, 1962.
- Engelbart, D. C. and English, W. K. "A Research Center for Augmenting Human Intellect." Proceedings, Fall Joint Computer Conf., 1968.
- Falor, K. "Software Forum-Survey of Program Packages-Programming Aids." Modern Data, March 1970.
- Fisher, D. A. Control Structures for Programming Languages. Ph.D. Dissertation, Computer Science Dept., Carnegie-Mellon U., Pittsburgh, Pa., May 1970. Published as Burroughs Corp. Report 64005 (TR70-C), May 1970.
- Green, J. "Program Analysis-A Problem in Man-Computer Communications." NASA Technical Report NASA TR R-338, June 1970.
- Grishman, R. "The Debugging System AIDS." Proceedings, Spring Joint Computer Conf., pp. 59-64, 1970.
- Hamming, R. W. "1968 ACM Turing Lecture." Computer Review, 11, 10, (cover), 1970.
- Jonsson, S. I. "On-Line Program Debugging." BIT, 8, pp. 122-127, 1968.
- Joyce, J. D. and Cianciolo, M. J. "Reactive Displays: Improving Man-Machine Graphical Communication." AFIPS Conference Proceedings pp. 713-722, Thompson: Washington, 1967.



- Knuth, D. E. An Empirical Study of Fortran Programs, Stanford U. Computer Science Department, Report #CS-186, 1971.
- Kocher, W. "A Survey of Current Debugging Concepts." NASA Contractor Report, NASA CR-1397, Aug. 1969.
- Ledgard, H. F. "Ten Mini-Languages: A Survey of Topical Issues in Programming Languages." Computing Surveys, 8, (3), 115-164, 1971.
- Lettvin, J. Y., Maturana, H. R., McCulloch, U. S., and Pitts, W. H. "What the Frog's Eye Tells the Frog's Brain." Proc. IRE, 47, pp. 1940-1951, 1959.
- Liskov, B. H. and Towster, E. The Proof of Correctness Approach to Reliable Systems. The MITRE Corp. ESD Cont. #F19(628)-71-C-0002, Project 6710, 1971.
- Luce, R. Duncan Individual Choice Behavior. New York: Wiley, 1959.
- Magunsson, D. A Study of Ratings Based on T.A.T. Stockholm: Almquest & Wicksell, 1959.
- Magunsson, D. Test Theory. Stockholm: Almquest & Wicksell, 1966.
- Miller, G. A. "The Magical Number Seven, Plus or Minus Two." Psychol. Rev., 63, pp. 81-97, 1956.
- Miller, J. C. and Maloney, C. J. "Systematic Mistake Analysis of Digital Computer Programs." Comm. ACM, 4, No. 6, 1963.
- Mills, H. D. "Syntax-Directed Documentation." Comm. ACM, 13, No. 4, April 1970.
- Mills, H. D. "Structured Programming in Large Systems." Debugging Techniques in Large Systems, Prentice Hall, New York, (forthcoming).
- Newell, A., Freeman, P., McCracken, D., and Robertson, G. (in press) "The Kernel Approach to Building Software Systems." Computer Science Research Review. Publisher unknown.
- Newman, J. R. Extensions of Human Capabilities Through Information Processing and Display Systems. System Development Corp., professional paper SP2560/000/00, 1966.

- Overton, R. K. "Some Data and Comments on Brain and Computer Memory Capacities." Proceedings, San Diego Symposium on Biomedical Engineering, 1961.
- Overton, R. K. "Manual and Automatic Interpretation of Near-Threshold Visual Events." Proceedings, Sixth Annual Symposium on Biomathematics and Computer Sciences, Houston, 1968.
- Rice, R. and Smith, W. R. "SYMBOL - A Major Departure from Classic Software Dominated van Neumann Computing Systems." Proceedings Spring Joint Computer Conf., AFIPS, pp. 575-587, 1971.
- Rosin, R. F. "Supervisory and Monitor Systems." Computing Surveys, 1, No. 1, March 1969.
- Smith, Karl U. Cybernetic Principles of Learning and Educational Design. New York: Holt, Rinehart & Winston, 1966.
- Smith, W. R., Rice, R., Chesley, G. D., Laliotis, T. A., Lundstrom, S. F., Calhoun, M. A., Gerould, L. D. and Cook, T. G. "SYMBOL - A Large Experimental System Exploring Major Hardware Replacement of Software." Proceedings, Spring Joint Computer Conf., AFIPS, pp. 601-616, 1971.
- Tallard, George A. and Waugh, Nancy C., Eds. The Pathology of Memory. New York: Academic Press, 1969.
- Telford, Charles W. and Saurey, James M. Psychology. Belmont, Calif.: Brooks/Cole, 1968.
- Vander Noot, T. J. "Systems Testing -- a Taboo Subject?" Datamation, 17, (22), 60-64, 1971.
- Wegbreit, B. The ECL Programming System, Division of Engineering and Applied Physics, Harvard University, Cambridge, Mass., 1971.
- Wells, Herbert "Subject - Controlled Inter-Trial Intervals in Concept Learning." Psychonomic Science, 19, (2), 109-110, 1970.
- Wersan, S. BALGOR, A Program for Solving Linear Programs in Zero-One Variables. Rpt. No. TR-0158(S816-10) 1, Aerospace Corp., San Bernardino, 1967.
- Wirth, N. and Weber, H. "EULER: A Generalization of ALGOL, and its Formal Definition: Part I." Comm. ACM, 9, no. 1, Jan. 1966.
- Wolin, B. R. "Complex Behavior in a 'Simple Task'." Second Annual Human Factors Symposium. USC: Los Angeles, 1962.
- Yntema, D. B. "Keeping Track of Several Things at Once." Human Factors, 5, pp. 7-17, 7, 1963.

unclassified

Security Classification

## DOCUMENT CONTROL DATA - R &amp; D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Corporation for Information Systems Research and Development (CIRAD), 401 N. Harvard, Claremont, California 91711		2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	
		2b. GROUP N/A	
3. REPORT TITLE A STUDY OF FUNDAMENTAL FACTORS UNDERLYING SOFTWARE MAINTENANCE PROBLEMS: FINAL REPORT			
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) None			
5. AUTHOR(S) (First name, middle initial, last name) None			
6. REPORT DATE December 1971		7a. TOTAL NO. OF PAGES 142 + Appendices	7b. NO. OF REFS 54
8a. CONTRACT OR GRANT NO. FI9628-71-C-0125		9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-72-121 Vol. 1	
b. PROJECT NO.			
c.			
d.		9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
10. DISTRIBUTION STATEMENT Approved for public release; distribution unlimited.			
11. SUPPLEMENTARY NOTES		12. SPONSORING MILITARY ACTIVITY Deputy for Command and Management Systems, Hq Electronic Systems Division (AFSC), L G Hanscom Field, Bedford, Mass. 01730	
13. ABSTRACT "Problems faced by programmers who must maintain programs someone else wrote" were identified. They were reduced to three fundamental inhibiting factors: (1) the limited rate at which people can make "relevance tests," (2) over-confirmation in clues required before hypothesis-testing, and (3) human vulnerability to distraction and procrastination. Studies suggested collectively by these factors were conducted. The studies (1) ascertained that programmers tend to think in terms of conceptual groupings whose objective identification would be helpful, (2) indicated that it was feasible to trace the path the programmer takes as he prepares to make a modification, and (3) identified a few tentative measures of the degree of maintainability of computer programs.			

DD FORM 1473  
1 NOV 65

Security Classification

14.

KEY WORDS

LINK A

LINK B

LINK C

ROLE

WT

ROLE

WT

ROLE

WT

computers

programmers

maintenance programming

documentation

human engineering

problem-solving

software systems

program modification

perception

program organization





